MASTER

Model-based verification of baggage handling control software

Arends, P.A.

Award date:
2016

Disclaimer
This document contains a student thesis (Bachelor's or Master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
Model-based verification of baggage handling control software

P.A. Arends
Thursday 25th August, 2016
Model-based verification of baggage handling control software

MASTER OF SCIENCE THESIS

For obtaining the degree of Master of Science in Business Information Systems at the department of Mathematics and Computer Science of Eindhoven University of Technology.

TU/e

VANDERLANDE

P.A. Arends – 0717208
philiparends91@gmail.com

Thursday 25th August, 2016

Supervisors:
dr.ir. D.A. van Beek
prof.dr.ir. J.F. Groote
dr.ir. J.A.W.M. van Eekelen
ir. L. Swartjes

Eindhoven University of Technology
Eindhoven University of Technology
Vanderlande Industries
Eindhoven University of Technology
Summary

Vanderlande Industries is the worldwide market leader for delivering baggage handling systems (BHSs) for airports. In order to stay ahead in this competing market, they are constantly looking for new ways to reduce the total time needed to develop control software of baggage handling systems.

A promising development is model-based design (MBD) of baggage handling controllers. Using MBD, controller errors can be detected in early stages of the design cycle. Partly this is due to the (visualized) simulation and model-based verification possibilities. Errors found in this early stage can still be resolved quickly. Recent projects have proven that controllers of BHSs can be made using this approach. The CIF tooling was used to model these controllers.

The goal of this project is to find an approach for verifying models of BHS controllers that are designed using CIF. The CIF tooling has no direct support for hybrid verification. Therefore, the CIF models are transformed using an available transformation tool. However, this tool does not support continuous variables. A method is developed that discretizes the hybrid model into an abstracted model. As these abstracted models are shown to have a too large state-space, a reduction of state-space is needed. Therefore, a method is applied that reduces a model w.r.t. the properties to be verified.

A hybrid CIF model for a controller of a BHS that consists of one conveyor is used to show that the approach developed in this project can be used for verifying properties. Three properties are verified for this model. A method for verifying a model consisting of more conveyors is given - but due to time limitations - not applied.
# Contents

## Summary

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.2 Preceding work</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1.3 Model checking tools</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>1.4 Outline</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Problem description</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2.1 Problem</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2.2 Approach</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Method for abstracting hybrid CIF conveyor models</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>3.1 Transformation tools</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>3.2 Possible abstraction methods</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>3.3 Chosen abstraction method</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>3.4 Implementation and Specification</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>3.5 Modelling of the CIF urgency concept</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>3.6 Over-approximating the hybrid model</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>A hybrid conveyor model</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>4.1 Choosing a model</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>4.2 Functionality</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>4.3 Validation and visualization</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>Abstracting and verifying the hybrid model</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>5.1 Requirements</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>5.2 Properties</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>5.3 Implementation</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>5.4 Specification</td>
<td>44</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>6.1 Conclusion</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>6.2 Recommendations</td>
<td>50</td>
</tr>
</tbody>
</table>
A Correcting the CIF to mCRL2 transformation tool 53
B Explaining some modal $\mu$-calculus notions 61
C Hybrid CIF model UFO sorter 65
D Abstracted CIF model UFO sorter 105
E CIF specification of single conveyor UFO sorter model 139

Bibliography 149
Chapter 1

Introduction

In recent years, the demand for large industrial automation systems has grown [1]. Additionally, the size of these systems has grown. This results in an increasing complexity of industrial systems. Due to the increased complexity, an increasing amount of time is spent on designing and testing the control software for these systems [2].

Testing the correct behaviour of the control software of industrial automation systems is a time consuming and challenging task. As the size and complexity of software increases, so does the time needed to ensure the correct behaviour of the controller.

During the design process, requirements for the software often change. This means that numerous lengthy design and testing iterations are needed.

Current testing procedures are able to detect many software errors, but not all of them. Because of this, companies are looking for new methods for decreasing the number errors in their software. Additionally, they want to reduce the time needed to find those errors.

Model-based verification helps in quickly finding errors. It decreases the total time needed to design control software of high quality.

1.1 Background

In order to explain how model-based verification can decrease the total time needed to produce high quality control software, it is important to first understand conventional methods of validating control software.

A much used conventional method for producing high quality software is the so-called V-model [3]. This model is shown in Figure 1.1. The model specifies the parts to deliver and in what order. The left part of the ‘V’ ensures that the software is defined in a
correct way. After implementing the proposed software design, the right part of the ‘V’ ensures proper validation and verification of the software.

Requirements for controllers of industrial systems are usually written down in an informal manner. They consist of a series of words describing what the system should be able to do. They are specified by a domain engineer, after consultation with a client. These requirements are then interpreted by a software engineer who writes the control software for the system.

Many of these requirements can be interpreted in different ways by the software engineer. These differences in interpretations are sometimes only found during the tests. At that moment, changing the design of the system and testing the system again is a time-consuming and therefore expensive process. It can even happen that the whole V-model design process needs to start all over again. These problems and their impact can be reduced by using a model-based design approach.

First, the advantage of using a model-based design approach for designing software is that it is less complex to design compared to directly programming the low-level code itself. Second, it is easier to interpret for people with different backgrounds (e.g. mechanical engineers, software engineers). Third, the model can be transformed automatically to different types of low-level controller code. Fourth, adapting a model when requirements change can be done faster than adapting the low-level controller code. Last, the (visual) simulation and verification options available in model-based design tooling reduces the number of real life system tests that are necessary for a correct functioning system. These advantages help to decrease the number of times the whole V-model needs to be repeated.

**Figure 1.1:** The V-model [3] is a linear software development method for developing and verifying (control) software.
again. This reduces the total design time for a correctly functioning controller. In [4], evidence is given that the model-based design approach can help in improving system quality and in decreasing the total design time.

1.1.1 Verification

In order to test requirements of the system, properties can be formulated. This requires the requirements to be translated to a detailed formal system design using a formal design language (e.g. automata).

A property is a logical expression which is unambiguous in its meaning. Therefore, they reduce the chance for misinterpreting the property. A property is used to test whether a requirement (or multiple requirements) are correctly reflected in the designed model. Properties ask questions about the detailed formal system design. They specify the states that should be reachable in the state-space and the values which the variables in those states should have.

Combing all the possible states a model can have together form the state-space of the model. Often the definition Labelled Transition System (LTS) is also used instead of state-space. Model-checking tools calculate (parts of) the state-space for verifying properties.

Model checking tools are able to read properties and verify whether or not a property holds on the formal system design. They can verify if a property holds for every possible state of the formal system design. Additionally, they are able to give a counter-example if a property does not hold. In order to achieve this, the control software needs to be designed using a (formal) model-based approach.

1.1.2 CIF

One way for designing a model-based controller is by making use of hybrid automata. Recent projects (see Section 1.2) have proven that it is possible to design model-based baggage handling controllers using the Compositional Interchange Format (CIF) [5] modelling language.

CIF is a hybrid automata based modelling language in development at the Systems Engineering Group of the Mechanical Engineering department of the Eindhoven University of Technology. The most recent version is CIF3.

Using CIF, it is possible to specify discrete event, timed and hybrid systems. The CIF tooling supports the entire development process of controllers, e.g. automata specification, supervisory controller synthesis, simulation-based validation and visualization, real-time testing, and code generation.
Throughout this thesis, the CIF language will be used for designing controllers of baggage handling systems.

For verifying properties of a model, a model-checking tool must be used. The CIF tooling has no built-in support for model-checking. However, CIF is able to translate CIF models to different types of model-checkers using several available transformation tools.

1.1.3 Vanderlande

Vanderlande is a manufacturer of automated material handling systems. It is the global market leader in baggage handling systems for airports, and sorting systems for parcel and postal services. The company is also a leading supplier of warehouse automation solutions.

The company is a global player with its headquarters in Veghel, The Netherlands. The total order intake over a 9 months period in 2015 was 1,16 billion Euro, making it the fifth-largest material handling systems supplier [6].

The company makes unique material handling systems for each of its customers. However, they try to use standardized components as much as possible. In this competitive industry, they seek for ways to decrease the total time-to-market of their systems.

Clients from different regions have preferences for different hardware controllers. These hardware controllers are called Programmable Logic Controllers (PLCs). These PLCs each use different programming languages. Currently, employees have to manually rewrite code for the different programming languages. The company would benefit from automatic code generation to different PLC languages.

Vanderlande uses similar testing procedures as mentioned in the beginning of Section 1.1. This means that domain engineers specify the requirements, which the software engineers interpret and implement. Finally, testing of these requirements takes place. Making a CIF model of the system and specifying properties can greatly reduce the time necessary for testing their systems.

1.1.4 Baggage handling systems

A baggage handling system (BHS) is a system responsible for the transport of baggage items. Usually, these systems are found at airports. They are highly automated systems, responsible for delivering baggage at their correct destinations (e.g. airplanes or baggage pick-up area’s).

Large BHSs are able to process thousands of products per hour. Important additional functions of BHSs are screening, sorting and temporary storage of bags. In Figure 1.2, part of a BHS can be seen. In this figure, a product is sorted onto another conveyor.
1.2 Preceding work

This graduation project is part of a collaboration between the Eindhoven University of Technology and Vanderlande. This collaboration takes place in the form of a PhD project of Lennart Swartjes, called the SUCCESS project. In this project a new way of developing control software is proposed. Instead of directly programming the control software, a controller model is developed, based on the requirements. From this model, the actual control software can automatically be generated.

A project overview of the SUCCESS project can be seen in Figure 1.3. The first three years of the project were mainly contributed to automatic PLC code generation and implementation. More about this topic can be read in the graduation projects of Rik Kamphuis [7], Sjors Jansen [8] and Tom Zwijgers [9]. Additional research topics are validation by interactive simulation and visualization and supervisory control synthesis.

In this graduation project, verification of CIF models for BHSs of Vanderlande is the topic of research. It is part of the sixth and final subject of the SUCCESS project.
1.3 Model checking tools

Tools for designing and verifying formal models are in development for over 25 years already [10]. Currently, many tools exist which can help an engineer design and verify complex system models. Well known tools are mCRL2 [11], UPPAAL [12], NuSMV [13] and Dezyne [14]. These tools are discussed below. In order to verify properties on a CIF model, one of these tools should be used.

mCRL2 is a formal specification language with an associated toolset. The toolset [15] can be used for modelling, validation and verification of concurrent systems and protocols. mCRL2 uses a process algebra language in order to describe the behaviour of systems. The tool-set is founded by Jan Friso Groote and is currently developed by the Formal
Systems Analysis group at the Eindhoven University of Technology. A transformation tool that transforms a subset of CIF concepts to mCRL2 is available [16].

UPPAAL is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata, extended with data types. A transformation tool that transforms a subset of CIF concepts to UPPAAL is available [17].

NuSMV is a reimplementation and extension of the SMV symbolic model checker, the first model checking tool based on Binary Decision Diagrams [18] (BDDs). The latest version of NuSMV is capable of SAT-based model checking [19] as well. The tool is aimed at reliable verification of industrially sized designs. A transformation tool that transforms a subset of CIF concepts to NuSMV is currently in development by Lennart Swartjes.

Dezyne is a commercial model-driven software engineering tool that enables software engineers to create, explore and formally verify designs for state based, event driven or concurrent software systems. It is the successor of the tool ASD [20][21].

1.4 Outline

In Chapter 2 a problem description for this thesis and an approach to research this problem are given. Afterwards, a suitable method for abstracting hybrid CIF models is given in Chapter 3. In Chapter 4, a hybrid conveyor model designed in CIF is explained and some modifications to this model are made. This conveyor model is abstracted and some properties for this model are verified in Chapter 5. Finally, conclusions and recommendations about the abstraction and verification results are given in Chapter 6.
Problem description

In this chapter a problem description for this thesis is given and an approach to investigate this problem is elaborated on.

2.1 Problem

Verification of properties is an important step in producing high-quality control software. The model-based design approach makes verification of properties possible. Using model checking tools it is possible to verify these properties. However, models for baggage handling systems contain both continuous time and discrete behaviour. The considered model checking tools only support discrete behaviour. Therefore, some kind of abstraction is needed.

An important problem throughout this thesis is how the properties for CIF baggage handling control software can be verified using model-based verification. This problem is made more clear by dividing it into two main research questions:

- ‘How can the hybrid model be abstracted?’
- ‘How can the abstracted model be used for verifying properties?’

On an abstracted model, properties can be verified. In order to be sure that the properties also hold on the hybrid model, it is essential for the abstracted model to behave ‘similar’ as the hybrid model.

The models of baggage handling control software of Vanderlande are relatively large models. Model checking tools are known to have problems verifying large models (the so called ‘state-space explosion’ problem [22]). The abstracted model therefore needs to be small enough such that the model checking tool is still able to verify the properties.
2.2 Approach

In order to answer the research questions from Section 2.1, an approach is used that starts by taking a very small discrete conveyor system. Using the mCRL2 and NuSMV transformation tools, this system can be transformed to the language of the model checker. After this, it is possible to verify properties for this system. This completes a first model-verification cycle. The goal of this first cycle is to get familiar with the tools and to find unexpected problems in an early stage.

After successful completion of the first cycle, the next step is to verify an abstracted model of a bigger hybrid conveyor system. Once that model is successfully verified, another cycle will start with a hybrid conveyor model containing even more functionality. At each cycle, the functionality of the hybrid model increases. Using this method, many size-related problems can be found at an early stage.

In Figure 2.1, an overview of the approach used for each cycle is shown. In this thesis, only the results for the last cycle (the biggest conveyor model) are reported on.

2.2.1 Hybrid model

A hybrid model is always designed according to some set of requirements. In this thesis, already available CIF conveyor models are used. In order to verify some more interesting properties of a chosen model, it is possible that this model still needs to be adapted slightly. In order to check the adapted hybrid model for errors, the CIF (visualization) simulator can be used. If errors are found during this simulation, the models must be adapted again. When the modeller has run enough simulations and feels confident of the correct functioning of the model, the abstraction phase can start.

2.2.2 Abstraction

For every hybrid model, it is necessary to find a suitable abstraction method. This is a challenging task. The problem is approached by searching for available literature about this topic. A best method for abstracting hybrid CIF models then needs to be chosen and applied. It is likely that the chosen method needs some additional adjustments to deal with the urgency concept of the CIF language.

The abstracted model must be ‘similar’ - with respect to the properties to be verified - as the hybrid model. Therefore, the behaviour of the abstracted model is compared to the hybrid model by means of simulation and visualization. It is possible that the modeller still finds some differences between both models, the abstract model then needs to be adjusted. Once the modeller is confident that the behaviour of the abstracted model is ‘similar’ to the hybrid model, the verification phase can start.
2.2.3 Verification

Once an abstracted model is made, the verification of properties can start. Because the model-check languages are different from the CIF language, they need to be transformed using the available transformation tools. The properties can then be verified using these model-checkers.

If a property does not hold, the counterexample that is given can be used to check whether indeed the property also does not hold on the hybrid model. If the property does also not hold for the hybrid model, the hybrid model needs to be adjusted. If the counterexample does hold on the hybrid model, then the counterexample is a spurious counterexample, and the abstracted model needs to be adjusted.

If a property does hold on the abstracted model, it also holds on the hybrid model (under the assumption that the abstracted model is a correct abstraction of the hybrid model).
**Figure 2.1:** An overview of the approach used throughout this thesis.
Chapter 3

Method for abstracting hybrid CIF conveyor models

In this chapter the search towards a suitable method for abstracting hybrid CIF conveyor models is conducted. First, the available transformation tools for verifying CIF models are mentioned. Second, a small literature study is conducted in order to find already documented abstraction methods for designing abstracted models. Third, a most suitable abstraction method is chosen. Fourth, the ‘Implementation and Specification’ method is explained. Fifth, the urgency concept of CIF is abstracted. Last, a method is proposed for testing whether the abstracted model is an over-approximation of the hybrid model.

3.1 Transformation tools

Model checking tools (see Section 1.3) can be used for verifying properties of modelled controllers. In the SUCCESS project, modelled controllers are designed using the CIF language. The model checking tools use a different modelling language than the CIF language. Therefore, the CIF models need to be transformed to a model check language. For this purpose, the following transformation tools are already available:

- CIF to UPPAAL tool \(^1\).
- CIF to NuSMV tool.
- CIF to mCRL2 tool \(^2\).

All three tools can be used for verifying properties for modelled controllers. Each transformation tool and corresponding model checker has its advantages and disadvantages.

\(^1\)http://cif.se.wtb.tue.nl/tools/cif2uppaal.html
\(^2\)http://cif.se.wtb.tue.nl/tools/cif2mcrl2.html
The most important differences are found in the expressiveness of the language used for verifying properties, and in the time needed to verify properties of large systems.

The NuSMV tool is still in development and can therefore not (yet) be found on the systems engineering Wiki page.

NuSMV can handle both the full sets of LTL and CTL logic. Additionally, NuSMV can give a counterexample if a property is not satisfied. For CTL, this is not always possible.

UPPAAL is able to verify a system with properties specified using a subset of CTL temporal logic.

mCRL2 uses \( \mu \)-calculus to specify properties on the system. Using mCRL2, much more complex properties can be verified in comparison to UPPAAL and NuSMV. mCRL2 also has the possibility to give a counter-example, if a property is not satisfied. Unfortunately, because of internal transformations these counter-examples are difficult to trace back to the original system.

Because the mCRL2 and NuSMV model checkers are able to verify larger systems than UPPAAL, these two model checkers will be used for verifying properties.

### 3.2 Possible abstraction methods

Both the mCRL2 and NuSMV transformation tools have limitations on the CIF concepts they are able to transform. The most important one is that they do not support continuous behaviour, only discrete behaviour. The concept of time in CIF is not supported by these transformation tools. Thus, in order to verify a CIF model containing timed behaviour, an abstraction of this model needs to be made.

The problem of designing an abstracted model is not new at all. It is a topic of research for decades already. In literature, two distinguishing methods for designing abstracted models are identified:

- Discrete-time methods
  - E.g., fictitious-clock model
- Dense-time methods
  - E.g., timed automata

In order to show the different abstraction methods, both methods are used to abstract the hybrid automaton of Figure 3.1.
METHOD FOR ABSTRACTING HYBRID CIF CONVEYOR MODELS

\[ t := 0 \rightarrow S_1 \quad a \text{ when } t > 1 \quad i = 1 \]
\[ S_2 \quad b \text{ do } t := 0 \]

**Figure 3.1:** A hybrid automaton that will be abstracted using different abstraction methods.

### 3.2.1 Discrete-time methods

Discrete-time methods use integer-valued variables for registering time. Time is a monotonically increasing sequence of integers. The discrete time-method approximates a continuous time system by using a fixed time step.

An advantage of discrete-time methods is that they can easily be transformed to ordinary finite automata. Many verification tools exist that can analyse these discrete-time methods.

The disadvantage of these methods is that the hybrid model is only approximated in its behaviour. Therefore, a modeller is restricted in his/her capabilities of modelling real-time systems.

An example where this approximation could be a problem is the following: Suppose the sensor of a conveyor belt needs to record the time between a `sensor_on` and `sensor_off` event. For example, this could be necessary for the detection of ‘fake’ products, when an employee accidentally waves his/her hand across the sensor. The time between the `sensor_on` and `sensor_off` events is very small when this happens. It is likely that the smallest time step chosen in the approximated model is larger than the time between those two events. Therefore, the detection of ‘fake’ products on a conveyor is not possible using the chosen approximated model. Although it is possible to lower the size of the smallest time step in the approximated model and to still model the detection of ‘fake’ products, this is very likely to result in the state-space explosion problem when trying to verify properties for the model (this is more elaborated on in Section 3.3).

**Fictitious-clock method**

An example of a discrete-time method is the fictitious-clock model [23] (FCM). When the (continuous) time increases in the hybrid model, the FCM method adds an event `tick` to the existing trace, indicating that time has passed. If more time passes before a next event happens, more `tick` events are added to the existing trace. The number of `tick` events between two non-`tick` events indicates the amount of time that has passed. In this way, the timed trace is transformed to an untimed trace in conventional formal language.
The tick event is a synchronizing event. Synchronization means that all events in the model with the same label name can only be ‘fired’ together. Thus, if one of the events with the same label is unable to ‘fire’, none of the events (with the same label) are allowed to ‘fire’. Thus, if one of the tick events is unable to fire, time cannot increase.

As an example, the automaton in Figure 3.1 is abstracted using the FCM method. This abstracted version is shown in Figure 3.2. In this abstracted version, a tick event corresponds to a discrete time step of $\Delta t = 1$.

![Figure 3.2: A abstracted version of the automaton of Figure 3.1 using the FCM method.](image)

It is possible to reduce the size of the automaton in Figure 3.2 by introducing variables in the automata. This is shown in Figure 3.3. In this figure, the non-negative integer $t$ is introduced to count the discrete time steps. The variable is also used to reset the time (event b). The behaviour of this automaton is exactly the same as the one from Figure 3.2. The advantage of this adapted FCM method is its better readability.

![Figure 3.3: A abstracted version of the automaton of Figure 3.1 using the adapted FCM method.](image)

Hybrid models that are abstracted using the adapted FCM method can be directly modelled in CIF. For verification purposes, the abstracted model can then be directly transformed to UPPAAL, NuSMV and mCRL2 using the available transformation tools (mentioned in Section 1.3). This makes it possible to verify the model using three different model checkers.

### 3.2.2 Dense-time methods

In dense-time methods [24], time is considered to be a monotonically increasing continuous variable. The time variable is always positive and has no bounds.
An advantage of dense-time methods is that they model physical systems in a realistic way. Usually, timed approximations of real-time systems are not necessary (for systems containing behaviour modelled by complex differential equations, approximations might still be necessary). Real-time systems can be modelled more easily compared to discrete-time methods.

A disadvantage of these systems is that they are hard to analyse, as most verification tools require discrete sequences of events as input language.

Generating a state-space for dense-time methods is not obvious. Usually, an interval of the continuous time variables refers to a single state. This interval is taken in such a way that no transition can happen while the values of the continuous time variables remain in that interval. This interval is also often referred to as a region. Deciding how to partition the system into these regions is usually done using a variant of the region graph construction method [24].

A problem in calculating regions is that the complexity is exponential in the number of continuous variables used [25]. Therefore, for large systems it is often not possible to calculate the (region) state-space for these systems. Thus, verifying properties for large systems is often not feasible for dense-time methods.

**Timed automata**

A theory used most often for modelling dense-time systems is the theory of timed automata and timed formal languages.

A timed automaton [24] (TA) is a finite automaton that uses additional real-valued clocks in its locations and edges in order to produce time controlled sequences of events. Every clock in the automaton increases with the same speed. Clocks can be reset by transitions. Guards on transitions allow for comparisons with the values of these clocks and can enable or disable the transition.

Unlike in the CIF language, clocks cannot be updated using (other) clock values (e.g., $t := t + 1$ is not allowed for TA). Also, guards on transitions cannot compare clock values (e.g., $t + 4 \geq 2t$). Only time derivatives equal to 1 are easy to model (e.g., $\dot{t} = 1$). This could be a problem when trying to model conveyors running at different speeds (e.g., $\dot{v} = 1$ and $\dot{v} = 1.5$). Last, TA are not familiar with the notion of synchronizing events, as used in the CIF language.

As an example, the automaton in Figure 3.1 is abstracted using the TA method. This abstracted version is shown in Figure 3.4. The automaton is actually exactly the same as the hybrid automaton. This is because in every state of the hybrid automaton, the derivative of the variable $t$ is equal to 1, which is the standard option in TA.

A trace of a timed automaton is a sequence of event labels where every label has an associated time-interval indicating the time at which the event happened.
TA can be modelled using CIF, however none of the available transformation tools is able to translate the continuous variables in CIF (which represent the clocks of TA). Therefore, TA should be modelled in another modelling tool (e.g. UPPAAL). Of the three mentioned model-checking tools, only UPPAAL is able to directly verify models designed using TA.

\[
\begin{align*}
  & t := 0 \\ 
  & a \text{ when } t > 1 \\
  & b \text{ do } t := 0 \\
\end{align*}
\]

Figure 3.4: A abstracted version of the automaton of Figure 3.1 using the TA method.

### 3.2.3 Comparing methods

The advantages and disadvantages of the possible abstraction methods are summarized in Table 3.1.

Table 3.1: Summarizing the advantages and disadvantages of the two abstraction methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapted FCM</td>
<td>Easily transformed to ordinary automata.</td>
<td>Hybrid models can only be approximated.</td>
</tr>
<tr>
<td></td>
<td>Can be analysed by most model-checking tools.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Can be modelled using CIF</td>
<td></td>
</tr>
<tr>
<td>Timed Automata</td>
<td>Approximations of hybrid model often not necessary.</td>
<td>Hard to construct region graph for large models.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not supported by many model-checking tools.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be transformed using the available CIF transformation tools.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Synchronizing events not supported.</td>
</tr>
</tbody>
</table>

### 3.3 Chosen abstraction method

The most suitable method for designing an abstracted CIF model is the FCM method. Using this method, the original hybrid CIF model can be adjust by simply substituting
the continuous variables with discrete variables. The continuous variables can easily be approximated in the abstracted model by introducing extra *tick* events.

The TA method requires an abstraction of the hybrid model to be designed in another tool (e.g., UPPAAL), which makes it very hard to compare the abstracted model to the hybrid model.

Comparing both models is very important in order to validate whether the properties verified on the abstracted model, also hold on the hybrid model. Therefore, the FCM method is most suitable for abstracting hybrid CIF conveyor models.

The adapted fictitious-clock method can be implemented in a straightforward way. When simple differential equations are used in the hybrid model (e.g., of the form $\dot{x} = c$ with $c$ a constant integer value), the continuous variables can be transformed in a trivial manner.

See for example part of a hybrid model of a box automaton in Listing 3.1 and an abstracted model in Listing 3.2. The continuous equation $\dot{x}$ in the hybrid part is replaced with several events *e_step* in the abstracted part. In Section 3.2.1, this event was labelled as *tick*. Each event *e_step* will update the variable $x$ with a constant value STEP.

Event *e_step* is a synchronizing event in all automata that previously used time (in the hybrid model). This means that event *e_step* can only be taken if this event can be taken by all those automata. In all of those automata, the same step in time is taken.

The conditional assignments (*if*, *elif* and *else*) in Listing 3.1 are not supported by the available transformation tools. Therefore, for every (*el*)if statement a separate edge is created in Listing 3.2 in order to result in similar behaviour. Using this principle, the *else* statement can be translated to a guard that is true if variable $x$ is not between 0 and 26. However, because the variable $x$ is bounded between 0 and 28, the guard is translated to $x \geq 26 \text{ and } x < 28$. 
Listing 3.1: Part of the box automata of the hybrid UFO sorter model.

```plaintext
// The velocity of a box is a function of the position and the corresponding
// velocity of the current conveyor

location Moving:
equation x' =
    if x < 8 : a_v1 * 1.0
    elif x >= 8 and x < 14 : a_v2 * 1.0
    elif x >= 14 and x < 20 : a_v3 * 1.0
    elif x >= 20 and x < 26 : a_v4 * 1.0
    else 1.0
```

Listing 3.2: Part of the box automata of the abstracted UFO sorter model.

```plaintext
// The position of the box is a function of the corresponding velocity
// of the current conveyor

location Moving:
edge e_step when x < 8 do x := x + (STEP * av1);
edge e_step when x >= 8 and x < 14 do x := x + (STEP * av2);
edge e_step when x >= 14 and x < 20 do x := x + (STEP * av3);
edge e_step when x >= 20 and x < 26 do x := x + (STEP * av4);
edge e_step when x >= 26 and x < 28 do x := x + (STEP * 1);
```

For this method, it is important to choose a large step size (constant `STEP`). A small step size results in a large state-space. This is because any discrete step (`e_step`) adds a new state to the state-space.

An example of this is shown in Figure 3.5. In this figure, two models of a conveyor system are shown. Products in the top conveyor need three times less steps to cross a single conveyor compared to products of the bottom conveyor.

In the top conveyor, a product can have two different positions at a single conveyor. Suppose every conveyor is allowed to have a maximum of one product on top of it. With this assumption, the maximal number of position combinations of the four products in the system is $3^4 = 81$. This is because a single conveyor has two possible positions for a product, and the empty state. The actual number is a little less because overlapping products are not taken into account.

For the bottom conveyor, with similar assumptions, the maximal number of position combinations of four products in the system is $7^4 = 2,401!$ Again, the actual number is a little less because overlapping products are not taken into account.

In this way, it is shown that the number of steps make the state-space increase exponentially. Therefore, when making an abstracted model, it is crucial to keep the number of steps as low as possible. The smallest amount of steps possible, while still maintaining a good approximation of the hybrid behaviour, is different for every hybrid model. In Section 5.3.1, the minimal number of steps for the model of that chapter is determined.
Figure 3.5: Discrete models of a conveyor system where a low number of discrete steps is used (top conveyor) and a high number of steps (bottom conveyor). The state-space of the model of the bottom conveyor is much larger than that of the top conveyor.

By keeping the number of steps as low as possible, the size of the state-space of the abstracted model can be reduced.

Unfortunately, for systems containing many more conveyors, the state-space still increases exponentially. In order to deal with this state-space problem, the ‘Implementation and Specification’ technique described in Section 3.4 can be applied.

### 3.4 Implementation and Specification

In this section a technique to deal with the state-space explosion problem of larger systems is introduced. This technique is explained in Part III of the book of [11]. The method is explained in short in this section.

In order to verify important properties, not all actions or events in a model are always necessary. Most of the properties only ask questions about some of the behaviour of a model. Also, there are a lot of questions that a modeller is not interested in. Therefore, it is possible to make another abstraction of the abstracted model. This is called a specification from now on. This specification only contains the events that are needed for verifying a chosen (previously defined) property (or multiple properties). These events are called the external events. All other events can then be labelled as non-visible $\tau$ events (internal events). The original (larger) abstracted model is called the implementation from now on. A specification is thus a reduction of the state-space of the implementation, containing only those states necessary for verifying a chosen property (or multiple properties).

The main idea of a specification is that it behaves branching bisimilar to the implementa-
tion, with respect to a certain property. *Branching bisimilarity* between the specification and implementation means that not only the sequence in which the external events happen is always the same, but also the choice of possible next events is always the same.

Because the state-space of a specification is much smaller, multiple single-conveyor specifications can be connected in sequence to verify properties for a large conveyor system. An example of this is show in Figure 5.2.

It is also possible to make a specification of multiple specifications (e.g., a single specification for a conveyor zone where a zone consists of multiple conveyor specifications put in sequence). In the end, it is then possible to verify properties that could not be verified using the implementation, as the state-space of the implementation is too large.

An example from [11] of a small implementation is shown at the left side of Figure 3.6. A specification of that implementation is found at the right side of Figure 3.6. States 1 and 2 are branching bisimilar with state 4. State 3 is branching bisimilar with state 5.

When looking at Figure 3.6, the events a and c are the external events. The event τ is an internal event. The behaviour of the specification is (branching bi-) similar to the behaviour of the external events of the implementation.

Proving that implementation and specification are branching bisimilar to each other can be done automatically using the *ltscompare* tool of mCRL2. Using this tool it is possible to compare larger implementations and specifications as well.

An example of the state-space of an actual conveyor model implementation is shown in Figure 3.7. A specification is made for this implementation. This specification is shown in Figure 3.8.

In order to check the branching bisimilarity between both models, all events not necessary for verifying a chosen property are hidden (they are shown as τ events in both figures.) The events that remain are called the external events.

Both the LTSs of Figure 3.7 and Figure 3.8 are branching bisimilar. It is immediately visible that the specification has a much smaller state-space then the implementation. For larger implementations, the difference between the size of the LTSs of the implementation and the specification becomes much larger.

### 3.5 Modelling of the CIF urgency concept

An important concept in CIF models is the concept of urgency. Urgency means that time can only progress in the model when there are no urgent events that can be taken any
more. This concept does not exist in the NuSMV and mCRL2 model checkers. Therefore, this concept needs to be translated.

In [26], priorities are modelled using the modal $\mu$-calculus (also for a system of Vanderlande Industries). It is possible to model urgency using modal logic, however the option of validating the abstracted model using the CIF (visualized) simulator would be lost by using this approach.

In CIF, urgency can be translated to a single CIF automaton. This extra automaton is added after the model is abstracted using the adapted FCM method. This single CIF automaton contains only one location with a single self-loop edge $\text{step}$. The guard of this self-loop edge is composed of all other guards of the other automata in the model. If any of the other (non-$\text{step}$) events are able to ‘fire’, the guard of this automaton becomes false and event $\text{step}$ cannot ‘fire’. Because event $\text{step}$ is a synchronizing event, time cannot pass in the whole model, if this guard is false. Thus, the non-$\text{step}$ events are more urgent than the time event $\text{step}$.

The general idea for translating the urgency concept is visualized in Figures 3.9 and 3.10.
Figure 3.7: LTS of the implementation of a conveyor model.

Figure 3.8: LTS of the specification of a conveyor model.
In Figure 3.9 a simple CIF model consisting of two automata is given. Initially, the variables $t$ and $x$ are set to value 0. In order to take the first transition $b$, time must increase. This will happen, as initially there are no other events that can be taken.

In Figure 3.10 the automaton consisting of only one location will check all guards and locations to see if any event can be taken. If this is not the case then event $\text{step}$ can be taken and the time in the model is increased with $\Delta$ (which is similar to the variable $\text{STEP}$ mentioned earlier). This is the main idea of applying the concept of urgency in abstracted models.

For larger models, the expression in the guard of location 5 (in Figure 3.10), can become quite large. Instead of manually constructing the guard, this process can be (partially) automated. This is done by first linearising all automata.

The automata in Figure 3.9 can be linearised. Linearising automata means merging all automata into one big automaton that results in exactly the same behaviour. This can be done automatically in CIF using the linearise-merge\(^4\) option of the CIF to CIF transformer\(^5\) tool. This tool generates one automaton with one location with a lot of self-loops. Now all guards of all these self-loops are ‘glued’ together with an ‘or’ $\lor$ operator and after this a negation $\neg$ operator is put in front of it. This newly constructed guard is then the guard for event $\text{step}$, similar to the event $\text{step}$ of location 5 in Figure 3.10. An example of the resulting guard for a large model is shown in Appendix D.14 (Section ‘Tick automaton that simulates urgency’).

### 3.6 Over-approximating the hybrid model

In this section, methods are given for ensuring that properties validated on the abstracted model, also hold on the hybrid model.

Ideally, the abstracted model should be an over-approximation\(^27\) of the hybrid model. With this it is meant that at least all possible sequences of events of the hybrid model are also possible on the abstracted model. If this is the case, a property verified on the abstracted model will also hold on the hybrid model.

There are two ways in CIF to validate if the abstracted model is an over-approximation of the hybrid model:

- By comparing both models using simulation and visualization.
- By synchronizing and running both models in parallel.

\(^4\)http://cif.se.wtb.tue.nl/tools/cif2cif/linearize-merge.html
\(^5\)http://cif.se.wtb.tue.nl/tools/cif2cif/index.html
Figure 3.9: An example hybrid CIF model that uses urgency to progress time. Events with the same labels are synchronizing events.

\[
\begin{align*}
\text{step } & \text{do } t = t + \Delta \\
\text{start } & \rightarrow 1 \quad \text{b when } t > 1 \quad \rightarrow 2 \\
& \text{a} \\
\text{start } & \rightarrow 2 \\
& \text{b when } t > 1 \\
& \text{a} \\
& \text{start } \rightarrow 3 \\
& \text{c} \\
\text{start } & \rightarrow 4 \\
& \text{a when } x > 2 \\
\text{start } & \rightarrow 5 \\
& \end{align*}
\]

Figure 3.10: An abstracted version of the automata in Figure 3.9. The step events increase time when no other event is possible any more. Events with the same labels are synchronizing events.
3.6.1 Simulation and visualization

The first approach is the most simple approach. The CIF tooling has options for simulating and visualizing a CIF model. By doing this for both the hybrid and the abstracted model, the behaviour of both models is compared.

A labour-intensive but thorough comparison is made by simulating both models event-by-event and checking for differences between both models. This is possible using the interactive GUI input mode of the CIF simulator. Here, the user can decide which next transition to take from a list of possible next transitions.

A much less labour-intensive but also less thorough comparison is by using the automatic input mode of the CIF simulator. This option automatically chooses which transition to take and thus automatically simulates the model.

Simulation and visualization can already give a good first estimate if the behaviour of the abstracted model is similar to that of the hybrid model. During this step, differences that are found can be adjusted for.

In order to be even more certain of a correct over-approximation, the second approach should be used as well.

3.6.2 Synchronizing both models

The second approach is to synchronize all events of both models and by running both models in parallel.

An example of this approach is shown in Listing 3.3. Here, part of a conveyor model that models the entry conveyor, is shown. An entry conveyor ‘generates’ products that can enter the conveyor system. Both hybrid an abstracted automata are made that model an entry conveyor. A graphical representation of the code in Listing 3.3 can be seen in Figure 3.11.

In Listing 3.3, the automata of the abstracted model are labelled with an ‘A’ in front of it. The other automata are hybrid automata. Automata with the same name (e.g. automaton entry for the hybrid model and automaton Aentry for the abstracted model) use the same synchronizing events (in the just mentioned example these events are e_snd_prd and e_send_prod). In models containing more automata, this is done for all hybrid-abstract automata pairs.
Listing 3.3: Example CIF code for synchronising both a hybrid and an abstracted conveyor model for validating whether the abstracted model is an over-approximation of the hybrid model.

```cif
// Declaration of global (synchronizing) events
event e_snd_prd, e_send_prod;

// HYBRID model
import "../V1SimpleHybrid/entry.cif";
entry : Entry(e_snd_prd, e_send_prod);

// ABSTRACTED model
import "../V1SimpleAbstract/entry.cif";
Aentry : AbstractEntry(e_snd_prd, e_send_prod);
```

Figure 3.11: A graphical representation of the code in Listing 3.3.
If the abstracted model is a correct abstraction of the hybrid model, no deadlock should occur when running both models in parallel.

If a deadlock does occur, then this is due to the fact that the abstracted model is unable to ‘follow’ an event of the hybrid model. The abstracted model then needs to be adjusted.

The advantage of this method is that it quickly shows any incompatibilities between both models. By running the model for a long time, it is possible to say - with a high level of confidence - that the abstracted model is an over-approximation of the hybrid model. However, this does not mean the abstracted model is guaranteed to be an over-approximation of the hybrid model. It possible that there exists a sequence of events in the hybrid model which cannot be followed by the abstracted model. This is because after running both models for a finite period of time, it is possible that this sequence of events simply has not taken place yet. In order to guarantee an over-approximation, both models should be run for an infinite amount of time, which is not possible unfortunately. Therefore, the method should not be considered as an absolute guarantee for the over-approximation notion.
A hybrid conveyor model

In this chapter a suitable hybrid conveyor model is chosen that can be validated by verifying its abstracted version using a model checker. First, the functionality of the hybrid model is explained after which the model is validated by making use of a visualization of the model.

4.1 Choosing a model

In order to apply the abstraction technique described in Chapter 3, a suitable hybrid model needs to be chosen. The model needs to have functionality that is representative of an actual conveyor system used at Vanderlande. However, in order to control the complexity of the model, not all functionality of a real baggage handling system is required.

A suitable model has been made for the Master’s course ‘4C650 - Supervisory Control of Hybrid Systems’ given by D.A. van Beek. In this course, the assignment was to model a baggage handling system (containing hybrid behaviour) with UFO and missing detection functionality. This functionality will be explained later on. The model designed by the students with the highest grade for this assignment (R.H.B. Roelofs and B. Volmer) is used to apply the developed abstraction method.

The UFO and missing detection functionality of this model are adapted in order for the system to behave more realistic (although an actual implementation of UFO-missing detection at Vanderlande would still be different on quite some points). The original model contained three different types of conveyor controllers (first, middle and last conveyor controllers). These three types are merged into one universal conveyor controller, that can be used at all positions in the conveyor system. The adapted model can be found in Appendix C.
4.2 Functionality

The chosen model consists of four conveyors that transport products from an entry conveyor to an exit conveyor. A visualization of the chosen model is shown in Figure 4.1. The controllers of the conveyors make sure that only one product is allowed on top of a single conveyor at all times. As mentioned before, the controllers of the conveyors have UFO and missing detection functionality.

The UFO detection functionality detects whether an unknown product has been thrown on the conveyor.

A UFO error refers to the state where a product is detected underneath the sensor of a conveyor while there is no product expected by the controller of the conveyor at that time. When a UFO error occurs, the UFO product will be sorted out to a dedicated conveyor track after the exit conveyor (this special conveyor track is not modelled).

Because the speed of all belts is known it is possible to predict when the products should arrive at the sensors of the conveyors. Therefore, a conveyor controller can expect at which moment in time a product should arrive at the sensor.

An missing error can happen when a product is expected to arrive underneath the sensor of a conveyor, but it does not.

The missing detection functionality is used in conveyor systems of Vanderlande to detect missing products that might have fallen off the conveyor or that are stuck somewhere in the system. A missing error causes the conveyor to stop, including all previous conveyors.

When a conveyor is stopped due to a missing error, operator intervention is required, as a bag might have fallen of the conveyor or that are stuck somewhere. Pushing the black button above the conveyor (see Figure 4.1) restarts the conveyor. It is assumed an operator solved the problem before restarting the conveyor. Once restarted, the same product can trigger another UFO error. When the product hits the sensor, a UFO error occurs. However, this UFO error is then ignored. This can only happen at the conveyor where the missing error happened previously. This functionality prevents an operator from having to press the restart button again.

Due to an imprecise speed of the conveyor belt it can happen that the product arrives a little later or earlier than expected. This can trigger UFO and missing errors. To overcome this, an error is not generated when the product arrives within a small ‘window of opportunity’. This ‘window of opportunity’ is defined as a small window around the sensor of the conveyor.

If the product is before the sensor at the expected arrival time, but within the ‘window of opportunity’, no missing error is generated. The expected position is then updated to
the observed position. The observed position is known once the product is underneath the sensor.

If the product is observed earlier than expected, it is validated whether the distance between the expected and observed position of the product is smaller than or equal to this 'windows of opportunity'. If this is the case, the expected position is updated to the observed position, and no UFO error is generated.

![Figure 4.1: UFO sorter model visualization in which conveyor 1 is in state 'Missing Error'.](image)

### 4.3 Validation and visualization

In order to validate the system, it is necessary to trigger UFO and missing errors. This can be done by creating products with an incorrect arrival time. This is done by delaying the product or by giving the product a head-start when this product is produced at the generator. Although this is not implemented, another option for creating incorrect arrivals would be to modify the speed of a conveyor.

In the model, the expected position of every product on the conveyor is tracked and visualized by a green contour around the product (see Figure 4.2). The contour is not always aligned with the corresponding product. In Figure 4.2, the first product is ahead of its contour. This means that the controller of conveyor 1 expects that the front of the product is at position 2 (position information is shown at the red bar underneath the conveyors), while the product is actually at position 5. When this product arrives at the sensor of conveyor 1, a UFO error is triggered, as the conveyor did not expect a product to arrive at that moment in time.

In Figure 4.2, the generator has given a head-start to the first product. This simulates that the actual speed of the conveyor is higher than the controller expected. Because the gap between the expected arrival (contour) and actual arrival (the product) is larger than a pre-defined size, a UFO error is about to occur.
When the controller expects a product to arrive (then the front of the contour is at the sensor), but there is no product near the sensor, a missing error will occur, see Figure 4.3.

If the gap between the contour and the product is smaller than a pre-defined distance measure, the position of the contour is updated to the actual position of the product. This is done because the incorrect arrival of the product is within acceptable margins. In a real conveyor system this means that the speed of the conveyor belt is almost equal to the expected speed of the conveyor belt.

Because the incorrect arrivals are always detected by the first conveyor, no UFO and missing errors can happen at the other conveyors. In order to test the other conveyors, a black button (see the top-left of Figure 4.2, above the word ‘Time’) is introduced that drops a new product on the next conveyor in line (conveyor 2). Pressing the button therefore drops a product just before the sensor of conveyor 2. This simulates that a random product is thrown on the conveyor system.

The sorting of a UFO product is represented as a textual change of the exit conveyor (see Figure 4.1) from ‘No UFO’ to ‘UFO’. In a real conveyor system, UFO products should be sorted onto a different conveyor than non-UFO products. In order to simplify the
visualization, this sorting of products is represented using the ‘UFO’ and ‘No UFO’ text changes in the exit conveyor. The exit conveyor checks whether the ‘mark’ of an entering product is set to ‘UFO’. If this is the case, it displays the text ‘UFO’, otherwise it displays ‘No UFO’. In Figure 4.4 a UFO is near the exit conveyor. Therefore, the text ‘UFO’ is shown.

When a UFO is detected, it is coloured red. In Figure 4.4, the UFO product nearing the exit conveyor started as a green product. It was detected at the first conveyor as a UFO and after that continued as a red UFO product. Missing products do not change colour as the controller does not know where the missing product is.

Figure 4.4: UFO sorter model visualization in which a UFO is close to the exit conveyor. Therefore, the text ‘UFO’ is shown.

After running the hybrid model for many simulation runs, the UFO and missing detection functionality have been validated. In order to be more sure of the correct functioning of this hybrid model, an abstraction of this model will be made in order to verify whether the model functions correctly. This will be done in Chapter 5.
Abstracting and verifying the hybrid model

In this chapter, first the requirements to be tested for the model are defined. Second, the requirements are formalized into properties such that they can be verified. Third, the model is abstracted for these properties using the method described in Chapter 3. Fourth, the results of the abstracted model are discussed and the properties are verified for a single conveyor system. Last, a specification is made for the abstracted model in order to verify a larger conveyor system.

5.1 Requirements

In order to show how the approach of Chapter 3 can be used for verifying properties of baggage handling systems, an interesting requirement for the model defined in Chapter 4 will be formulated.

An interesting requirement is the following:

‘If a UFO product is detected, it is always sorted out correctly at the exit conveyor.’

The requirement is made more specific by dividing it into three sub-requirements:

1. ‘Pushing the ‘UFO drop’ button (which drops a UFO on the first conveyor) always eventually results in a UFO product sorted out at the exit conveyor’.

2. ‘A UFO product dropped on a conveyor always takes the UFO exit at the exit conveyor’.

3. ‘After a UFO is detected by the exit conveyor and before that UFO leaves the exit conveyor, no non-UFO product leaves the exit conveyor.’ This means that a UFO product never leaves the system as a non-UFO product.
5.2 Properties

The requirements from Section 5.1 are formulated in natural language. In this section, these requirements are formalized such that they can be verified.

Important events that are necessary to formulate those properties are listed in Table 5.1.

Table 5.1: Important events in the model necessary for formulating the UFO properties.

<table>
<thead>
<tr>
<th>Event</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_pc</td>
<td>A product enters the first conveyor.</td>
</tr>
<tr>
<td>u_a_push</td>
<td>A UFO product is dropped at a conveyor.</td>
</tr>
<tr>
<td>c_UFO_incomingEvent</td>
<td>The exit conveyor detects an incoming UFO.</td>
</tr>
<tr>
<td>leaveNonUFO</td>
<td>A non-UFO product leaves via the exit conveyor.</td>
</tr>
<tr>
<td>leaveUFO</td>
<td>A UFO product leaves via the exit conveyor.</td>
</tr>
</tbody>
</table>

The properties are formulated using the modal $\mu$-calculus. This logic is used because a future specification (see Section 5.4) is made in mCRL2 and mCRL2 can only verify $\mu$-calculus.

In order to explain the properties in this section, some notions of the modal $\mu$-calculus first need to be explained. The important notions (syntax) necessary for reading the three properties in this section are explained in Table 5.2. The syntax descriptions used are from [11]. Some examples that explain these notions in more detail can be found in Appendix B.

5.2.1 First property

The result of formalizing the first sub-requirement of Section 5.1 is given below:

$$[true^*.u_a_push]_\mu X. ( \neg c_{UFOincomingEvent} X \land (true) true )$$

It states that it is always ( [...] ) the case that when after a sequence of events ( true*. ) an u_a_push event happens, then there must always ( second [...] ) exist a finite trace of actions ( $\mu X. (...) ) other ( \neg ) than c_UFOincomingEvent. Additionally ( $\land$ ), in that finite trace of actions, there must always be a next event ( $\langle true\rangle$ ) possible which leads to a state where true holds (thus: every state). This means that in that finite trace, no deadlock is allowed to occur.

Because the trace of non-c_UFOincomingEvent’s must be finite and because no deadlock can occur in this trace, eventually a c_UFOincomingEvent must happen.

So in short, the property says that when an u_a_push is done, eventually a c_UFOincomingEvent happens.
Table 5.2: Explaining the modal $\mu$-calculus syntax that is needed for understanding the three properties in more detail.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle a \rangle \phi$</td>
<td>The diamond modality is valid whenever an a-action can be performed such that $\phi$ is valid after this a has been done. So, the formula $\langle a \rangle \langle b \rangle \langle c \rangle true$ expresses that a process can do an a followed by b followed by c. Expressing that after doing an a action both a b and c must be possible is done by the formula $\langle a \rangle ((b)true \land \langle c \rangle true)$. Expressing that after an a action no b is possible can be done by $\langle a \rangle \neg \langle b \rangle true$.</td>
</tr>
<tr>
<td>$[a] \phi$</td>
<td>The box modality is valid when for every action a that can be done, $\phi$ holds after doing that a. So, the formula $[a] \langle b \rangle true$ says that whenever an a can be done, an action is possible afterwards. The formula $[a]false$ says that whenever an a is done, a situation is reached where false is valid. As this cannot be, the formula expresses that an action a is not possible. Likewise, $[a][b]false$ holds when a trace a b does not exist.</td>
</tr>
<tr>
<td>$R_1.R_2$</td>
<td>The regular formula $R_1.R_2$ represents the concatenation of the sequences of actions in $R_1$ and $R_2$. For instance, $\langle a.b.c \rangle true$ is the same as $\langle a \rangle \langle b \rangle \langle c \rangle true$ expressing that a sequence of actions a, b and c can be performed.</td>
</tr>
<tr>
<td>$\mu X. \phi$</td>
<td>The minimal fixed point.</td>
</tr>
<tr>
<td>$\nu X. \phi$</td>
<td>The maximal fixed point. An effective intuition to understand whether or not a fixed point formula holds is by thinking of it as a graph to be traversed, where the fixed point variables are states and the modalities $\langle a \rangle$ and $[a]$ are seen as transitions. A formula is true when it can be made true by passing a finite number of times through the minimal fixed point variables, whereas it is allowed to traverse an infinite number of times through the maximal fixed point variables. In the example with a single a-loop below, the formulas $\mu X. \langle a \rangle X$ and $\nu X. \langle a \rangle X$ can only be made true by passing an infinite number of times through X and/or s. So, the minimal fixed point formula does not hold, and the maximal one is valid.</td>
</tr>
</tbody>
</table>

\[ \begin{tikzpicture} 
    \node (start) at (0,0) {$s$}; 
    \node (a) at (1,0) {$a$}; 
    \draw [->] (start) -- (a); 
\end{tikzpicture} \]
5.2.2 Second property

The result of formalizing the second sub-requirement of Section 5.1 is given below:

1. \( \nu X(t : Int = 0, n : Int = 0, u : Bool = false) \).
2. \([c_{pc}]X(t + 1, n, u) \land \)
3. \([u_a_{push}]X(t + 1, t, true) \land \)
4. \([\text{leaveUFO} \cup \text{leaveNonUFO}](X(t - 1, n - 1, u) \land \text{val}(t > 0 \land (n > 0 \lor \neg u))) \land \)
5. \([c_{UFOincomingEvent}](X(t, 0, false) \land \text{val}(n \approx 0 \land u) \land \)
6. \([\neg(c_{pc} \cup \text{leaveUFO} \cup \text{leaveNonUFO} \cup u_a_{push} \cup c_{UFOincomingEvent})]X(t, n, u) \land \)

Table 5.3: The variables used in the second UFO property and their definitions.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The number of products in the system between the UFO product and the exit conveyor</td>
</tr>
<tr>
<td>t</td>
<td>The total number of products in the system</td>
</tr>
<tr>
<td>u</td>
<td>A boolean indicating that a UFO product is dropped on the belt (u = true) or not (u = false)</td>
</tr>
</tbody>
</table>

As this property looks quite impressive at first, it will be explained in the paragraphs below. First a global overview of the property is given, after which are more detailed step-by-step explanation is given.

The second property of Section 5.1 is verified by counting the number of products that are in front of the UFO product. This number is stored in variable \( n \). Correct behaviour then means that the exit conveyor detects a UFO product after \( n \) times a product has left via the exit conveyor. In order to update the variable \( n \), the number of incoming and outgoing products are recorded as well. The variables used by the property are described in Table 5.3.

Below, the second property will be explained line by line. The numbers before the explanations below correspond to the line numbers of the second property.

1. The first line of the property shows the maximal fixed point operator \( \mu \). It indicates that all events from the second until the sixth line can be done infinitely often.

2. In the second line, an incoming product (denote by event \( c_{pc} \)) updates the total number of products \( t \) in the system.

3. The third line updates the variables when a UFO product is dropped on the first conveyor (indicated by event \( u_a_{push} \)). The total number of products in the system \( t \) is increased and the boolean \( u \) indicating that a UFO product is dropped on the conveyor is set to \( true \). The number of products in front of the UFO product
n is set to the value of t. This is because the UFO product is dropped on the first conveyor and all products in the system are then in front of the UFO product.

4. The fourth line describes what should happen when a product leaves the system. A product leaving the system is indicated by event leaveUFO for UFO products and event leaveNonUFO for non-UFO products. The boolean expression \( t > 0 \land (n > 0 \lor \neg u) \) expresses that a product is only allowed to leave when there are products in the system (\( t > 0 \)) and when there is at least one product in front of the UFO product (\( n > 0 \)) (or there is no UFO product at all in the conveyor system (\( \neg u \))). When these conditions are met, the total number of products in the system \( t \) and the number of products in front of the UFO product \( n \) are both decreased by one.

If there is no more product in front of the UFO product, first the event c_UFO-incomingEvent should happen, before another product is allowed to leave the conveyor. This is described by the fifth line.

5. The updates to the variables of process X in the fifth line are triggered when a UFO product is detected by the exit conveyor (event c_UFOincomingEvent) but only when the boolean expression \( n \approx 0 \land u \) evaluates to \( true \). The expression expresses that event c_UFOincomingEvent is only allowed to happen when there are no more products in front of the UFO product (\( n \approx 0 \)) and only when a UFO product has been dropped on a conveyor previously (\( u \approx true \)). If these conditions are met, the variable \( n \) is reset to \( 0 \) and \( u \) is reset to \( false \) again.

The property is used to verify the implementation of the abstracted model. In this implementation, there are many other events than those shown in Table 5.3. In order to determine what should happen when an event appears that is different from those in Table 5.3, the sixth line in the property is added.

6. The sixth line simply describes that if any event other than c_pc \( \cup \) leaveUFO \( \cup \) leaveNonUFO \( \cup u_a_push \cup c_UFOincomingEvent \) happens, the process X simply continues, without changing any variables. Without this line, the property will not evaluate to \( true \) as there exists no infinite trace with only the events mentioned in Table 5.3.

By verifying this property, it is checked whether the sequence of events in the implementation always follows the pattern of events described by this property.

5.2.3 Third property

The result of formalizing the third sub-requirement of Section 5.1 is given below:

\[ [true*.c_UFOincomingEvent.(\neg leaveUFO)*.leaveNonUFO]false \]

The pattern \( [R]false \) indicates that the sequence of events in \( R \) always (\( [] \)) leads to a state where \( false \) holds. \( False \) can never hold in a state. So, in other words: the sequence of events in \( R \) can never happen.
The sequence of events $R$ in this formula describes that after some arbitrary sequence of events (true*) a sequence of events that do not (¬) contain any leaveUFO can happen in between a c_UFOincomingEvent and a leaveNonUFO.

This means that it is impossible to do a c_UFOincomingEvent followed by a leaveNonUFO, without a leaveUFO in between.

This results again in the original requirement: ‘After a UFO product is detected by the exit conveyor and before a UFO product leaves via the exit conveyor, no non-UFO product leaves the exit conveyor.’

In Section 5.3.3, the three properties formulated in this section are verified for the abstracted model.

As the properties for the hybrid model are now known, the abstraction of the hybrid model can start. This is done in the next section.

### 5.3 Implementation

#### 5.3.1 Abstracting model

The hybrid model described in Chapter 4 is abstracted using the method described in Chapter 3. In this chapter, it is described how the continuous variables in the hybrid model can be replaced by discrete e_step events, while remaining approximately the same behaviour.

In this section, some additional abstraction details that are specific to the chosen hybrid model are explained. First the minimal number of steps possible for a product to go over a conveyor is determined, after which some changes to the product generator are discussed. Last, the concept of urgency is implemented in the abstracted model.

As explained in Chapter 3, the number of steps that a product needs in order to go over a conveyor needs to be as low as possible. By looking again at the hybrid model, the minimal number of steps is found to be two. The reason for this is explained in the next paragraphs.

In the hybrid model, important changes in state happen only when the sensor of a conveyor changes from off to on or vice-versa. These important changes need to be captured in the abstracted model as well.

The important changes in state happen when the position of a product switches from before the sensor to at the sensor or from at the sensor to after the sensor. In Figure 5.1, the first product is before the sensor and the second product is at the sensor. The third product is again before the sensor (of conveyor four). However, one could also see the position of the third product as after the sensor of conveyor three.
In order for a product to ‘appear’ at these steps, two steps are required to cross a single conveyor.

**Figure 5.1:** The possible positions for products on the abstracted model of the conveyor system. The first product is *before the sensor* of conveyor 1, the second product is *at the sensor* of conveyor 2. The third product is *before the sensor* of conveyor 4. However, the third product could also be seen as *after the sensor* of conveyor 3.

After using the abstraction method of Chapter 3, an additional modification to the generator is necessary in order to get a functional abstracted model.

The hybrid model contains a stochastic generator which generates delays for the contours of the products by sampling from a uniform distribution. Such uniform distributions are not supported by the transformation tools. Therefore, they are removed in the abstracted version of this generator.

Additionally, the stochastic generator uses a lot of timers/clocks which cannot be used in the abstracted model.

The generator is therefore simplified in the abstracted model. The abstracted generator sends a product whenever the next conveyor is willing to accept a new product. The generator then waits two time units, before sending another product. In this way, the abstracted generator closely resembles the pace in which products are send using the original stochastic generator. Although, the ‘similarity’ of the abstracted generator is close to the hybrid generator, its behaviour cannot be made exactly similar. This limits the possibility of having an over-approximation between the two models, as defined in Section 3.6.

After these modifications, the abstracted model is almost finalized. The only step left is to model the concept of urgency into the abstracted model. This is done in the way described in Section 3.5. After this last modification, the abstracted model is finalized.

A four-conveyor version of the abstracted model is shown in Appendix D. By simulating both the hybrid and abstracted versions, the behaviour of both models is compared. Both models show very similar behaviour in the simulation.
5.3.2 Abstraction results

The abstracted model of Section 5.3.1 consists of four conveyors. In order to reduce the size of the total model an one-conveyor version of this model is made. This requires no extra modifications as the conveyors already use universal controllers.

The one-conveyor version of the abstracted model results in a state-space of 25,034 states and 92,622 edges. Both the NuSMV and mCRL2 model checkers are able to calculate the state-space in a reasonable time (about 1 minute).

Calculating the state-space for a four-conveyor version takes very long (both CIF, mCRL2 and NuSMV are used) and after calculating more than 18 hours, the process is stopped. Most likely, the state-space is simply too large.

For the mCRL2 model checker, an additional problem caused the long calculation time. Before a state-space can be calculated, an linear process specification (lps) needs to be calculated. Calculating this ‘lps’ also took too long (more than 18 hours).

The long ‘lps’ calculation time is further investigated and during that process some errors in the CIF to mCRL2 transformation tool are found and fixed. A more detailed report of the problems found and their solutions can be seen in Appendix A. These solutions are implemented in an improved version of the mCRL2 transformation tool.

Unfortunately, the improved mCRL2 transformation tool did not decrease the ‘lps’ calculation time (in fact it even increased the ‘lps’ calculation time). Together with A. Hofkamp (who implemented the CIF to mCRL2 transformer) it became clear that the basis of the transformation tool needs to be redesigned completely in order to lower the amount of time necessary for calculating the ‘lps’ for large models. This is a sidetrack which is not part of this thesis, and this activity is picked up by M.A. Reniers.

In Section 5.4, a specification is made of the one-conveyor abstracted model in order to verify a larger conveyor system.

5.3.3 Verification single conveyor

The implementation of the single universal conveyor obtained in the previous section, is verified using the three properties from Section 5.2. All three properties are verified to hold.

5.4 Specification

In this section the ‘Implementation and Specification’ technique described in Section 3.4 is applied to the abstracted implementation. This technique can be used to verify a system with (many) more conveyors.
The main idea is to make a specification of a single conveyor that uses the universal controller. This specification can then be put in sequence in order to verify properties for a larger conveyor system. By reusing (putting in sequence) a universal specification it is not necessary to calculate the state-space of a very large implementation. Because the state-space of a single specification is very small, it still remains possible to calculate the state-space of many specifications put in sequence.

5.4.1 Abstraction

An abstraction of the implementation for a (single) universal conveyor is made and is shown in Appendix E. A visualization of this specification is shown in Figure 5.2. The hybrid universal conveyor is in fact just a buffer that can hold only one product at a time. The type of the product is either UFO or non-UFO. The event NonUFO is the event that signals an outgoing non-UFO product. It is simply a shorter notation of the event leaveNonUFO. Event push drops a UFO on the corresponding conveyor. An outgoing UFO is indicated by event leaveUFO. The event push is blocked at the second and third conveyor as the implementation currently only allows a maximum of one UFO product at the entire conveyor system. The event push synchronises with the event UFO of the entry conveyor.

![Diagram of hybrid universal conveyor](image)

**Figure 5.2:** By putting multiple implementations and specifications in parallel, a larger UFO sorter system can be verified as well. Events UFO and NonUFO are short-hand notations of events leaveUFO and leaveNonUFO respectively.

By using the mCRL2 tool ltscompare, a single universal specification (including the entry
and exit conveyors) is proven to be branching bisimilar with a single universal conveyor implementation (also including the entry and exit conveyor). By putting this specification in sequence, as seen in Figure 5.2, the UFO properties can be verified for a larger UFO sorter (for example with eight conveyors).

5.4.2 Abstraction results

A single universal conveyor specification results in a state-space of 68 states. As the state-space of this model looks very ‘spaghetti-like’, it is converted modulo branching bisimilarity for better readability (using the mCRL2 tool \texttt{ltsconvert}). The resulting state-space is shown in Figure 5.3.

In Figure 5.3, some interesting requirements are noticeable. For example it becomes quickly visible that an \texttt{u_a_push} is always followed by a \texttt{c_UFOincomingEvent}, which is the first requirement of Section 5.1.

\begin{center}
\textbf{Figure 5.3:} State-space of the specification for a single conveyor UFO sorter model. For better readability, the state-space is converted modulo branching bisimilarity.
\end{center}
5.4.3 Verify properties multiple conveyors

A larger conveyor system can be verified using the same properties as those from Section 5.2. This is left as future research due to the time constraints set for this thesis. By following this approach, it is concluded that successfully verifying properties for large conveyor systems is feasible.
Conclusion

In this chapter, a conclusion is drawn on the research that has been conducted during this project. Additionally, recommendations are given for future research.

6.1 Conclusion

A problem of verifying hybrid controller models of BHSs modelled in CIF, is that these models are relatively large. In general, hybrid systems can only be verified for small systems. Therefore, an abstraction of these large models needs to be made.

Unfortunately, the CIF tooling has no direct support for hybrid model-checking: The CIF model needs to be transformed to an equivalent model that can be read by a model-checker. Three transformation tools are available for this purpose. Unfortunately, these transformation tools do not support continuous variables.

An important problem solved in this thesis is how to make an abstraction of a hybrid model, such that this abstraction can be used for discrete verification using a model-checking tool.

The method used for abstracting the hybrid CIF model is an adapted version of the fictitious-clock model (FCM), which discretizes the continuous behaviour. This method is used because it is easy to use and straightforward.

The CIF concept of urgency is not supported by the available transformation tools. Therefore, the concept is incorporated into the abstracted model. A semi-automated approach for this is given.

It is important that the abstracted model is an over-approximation of (or equal to) the hybrid model. In this way, a property that is verified on the abstracted model, also holds on the hybrid model. In order to validate whether the over-approximation relation holds,
two methods for comparing the hybrid and the abstracted model are given: ‘Simulation and visualization’ and ‘Synchronizing events’. Although these methods give confidence that there is an ‘over-approximation’ relation between the two, the methods cannot fully guarantee that this is the case.

Additionally, the ‘Implementation and Specification’ method of [11] is used to reduce the state-space of the abstracted model. This reduction is realized by removing all behaviour not necessary for the verification of certain properties.

It is shown that the developed methods can be used for verifying properties on a hybrid conveyor model. To this end, the hybrid model is abstracted using the adapted FCM method. A specification of a single conveyor of this model is made. Using this specification, three properties are verified.

It is possible to verify a larger conveyor system, consisting of many more conveyors (e.g., eight conveyors). This can be done by putting the designed specification in sequence. Due to time restrictions, this is not tested. However, the specification and the properties are formulated with this extension in mind.

The approach given in this thesis shows the potential for finding and reducing the number of controller errors found in BHSs using the CIF and mCRL2 tooling. In the future, this approach can act as a foundation for developing safe and reliable control software for BHSs at Vanderlande Industries.

6.2 Recommendations

6.2.1 Extension of project

A logical extension of this project is to verify the properties of Section 5.2 on a conveyor model consisting of multiple conveyors. For this purpose, the universal conveyor specification of Section 5.4 may be put in sequence.

A specification that consists of multiple conveyors (e.g., a zone) can be used as a building block for verifying properties on larger systems. By using this technique of making a specification of multiple specifications, eventually a complete baggage handling system may be verified.

Another extension is to add more functionality (from the product-book of Vanderlande) to a single universal conveyor model.

The approach for verifying model-based conveyor systems still requires some manual steps. An improvement would be to automate these steps. This prevents mistakes being made in the approach and can greatly decrease the time needed to apply the approach. Most time can be won by automating the urgency translation steps and by the automatic
transformation of continuous variables to discrete time steps. Important in this last step is that the continuous variables are always bounded.

6.2.2 Tool improvements

Some small changes for the ‘CIF to mCRL2’ transformation tool are advised. One change is the support of updates in assignments (do if $y \geq 3$: $x := 0$ else $x := 1$). Another change is supporting the automaton name in the action labels (e.g., event Box.leaveUFO in CIF should become action Box_leaveUFO in mCRL2), this prevents the need for generating leaveUFO and leaveUFO2 events when two automata in CIF have labels with the same name.

It is important to improve the readability of counter-example traces found using the mCRL2 toolset. Currently, it is hard to directly replay these traces on the original model. A tool that could automatically replay these traces in a simulator would help in quickly resolving any errors found in the model.

6.2.3 Design models for verification

The ‘Implementation and Specification’ technique should be made more familiar amongst developers who want to verify properties for their models. An idea would be to create a special Wiki-page that explains how to make a specification (in CIF) for their model(s). Amongst the possibilities of verifying properties, designing a specification for an implementation gains the modeller a lot of insight in the correct functioning of his/her model.

Finally, the syntax of CIF allows a modeller a lot of freedom in modelling complex systems. Unfortunately, continuous variables used in hybrid models are less suitable for verifying large systems. Many requirements of a model, can be modelled by using discrete (timed) variables only. Modellers should be made aware which variables/syntax to use (and which not) when they want to verify properties for their models. If a modeller wants to verify properties for the system, he/she should start the very first design of the system with this purpose in mind. Thus, the modeller should try to avoid using syntax that is difficult to verify. An idea would be to set the option of activating a ‘restricted’ CIF environment (in Eclipse) for producing models that can immediately be used for verification purposes.
Appendix A

Correcting the CIF to mCRL2 transformation tool

In this chapter two problems in the CIF to mCRL2 transformation tool are addressed and a solution to these problems is proposed and implemented.

Incorrect variable transformations

After the transformation from CIF to mCRL2 of an abstracted simple conveyor model (not shown in this thesis) that still contains a deadlock, a requirements check was done to see if mCRL2 was able to give a trace to the deadlock state. A formula to check for deadlock freedom was applied and the model appeared to be deadlock free, while it should not. The same model, transformed to NuSMV and tested on deadlock-freedom using NuSMV did result in a trace to a deadlock state. Also, simulation of the CIF model resulted in a deadlock. After consolidation with the creators of the transformation tool (dr.ir. A.T. Hofkamp, dr.ir. M.A. Reniers and dr. E.P. Vink) it turned out that this was caused by an extra self-loop that was added to the mCRL2 model for every variable in the CIF model. For example, a CIF automata of a discrete sensor can be seen in Figure A.1.
In the mCRL2 model, at every moment it is always possible to check the value of the variable $s$. A LTS of the mCRL2 process can be seen in Figure A.2. Because in mCRL2 there is no possibility to store variables, in this way it is still possible to read the current value of a variable. The downside is that even at a deadlock state, there is still a possibility to do one of these self-loops. This caused the deadlock-freedom requirement to hold. In the mCRL2 process instantiation, it can be specified not to allow these ‘value_’ self-loops. The downside is that the value of a variable cannot be checked any more by an observer from outside. After removal of these self-loops, the deadlock freedom requirement was expected not to hold any more. However, the requirement still returned the model to be deadlock-free. Therefore, another reason must be prohibiting a deadlock in the system.

Further investigation addressed another problem in the transformation tool. In order to explain the root cause of this problem, it is necessary to explain the CIF to mCRL2 tool in more detail.

Every variable in a CIF model that is used (read) by multiple automata will get a
separate process instantiation in the mCRL2 model. This separate variable process, called ‘VarProc’ is used by the processes of the automata, called ‘BehProc’ to read and/or write variables.

As an example, the CIF model of Figure A.3 is created. The value of the variable \( x \) is stored in the mCRL2 process shown in Listing A.1. This process takes care of ‘storing’ the correct value of variable \( x \). Both automata A and B can read this value. This is done by communicating with the action \( \text{vread}_x \) in process ‘VarProc\_x’. Automaton A can also write to variable \( x \) by communicating with action \( \text{vread}_x \). For communication, the automata use the variable \( \text{aread}_x \) or \( \text{awrite}_x \) in order to read or write the variable \( x \) in the ‘VarProc\_x’ process. The mCRL2 code only allows \( \text{vread}_x \) to occur when also an \( \text{aread}_x \) occurs. The same holds for \( \text{vwrite}_x \) and \( \text{awrite}_x \). In this way variable \( x \) is constantly kept up-to-date. The whole mCRL2 code of the automata in Figure A.3 can be found in Listing A.7.

![Automata A and B](image)

**Figure A.3:** CIF example for CIF to mCRL2 transformation purposes.

**Listing A.1:** mCRL2 process ‘VarProc\_x’

```plaintext
proc VarProc\_x(v: Int) =
    value\_x(v) . VarProc\_x(v) +
    vread\_x(v) . VarProc\_x(v) +
    sum m: Int . ((m >= 0) && (m <= 6)) -> vwrite\_x(m) . VarProc\_x(m) +
    sum m: Int . ((m >= 0) && (m <= 6)) -> vread\_x(v) | vwrite\_x(m) . VarProc\_x(m);
```
A problem in the transformed mCRL2 code is found at lines 5-6 in Listing A.2 in the communication operator. Here a multi-action \( \text{aread}_x \mid \text{vread}_x \) and \( \text{awrite}_x \mid \text{vwrite}_x \) will continue as if they are a single \( \text{aread}_x \) and \( \text{awrite}_x \) action respectively. Because single \( \text{aread}_x \) or \( \text{awrite}_x \) actions are still allowed and not affected by the communication operator, the following problem can occur:

- A multi-action \( \text{awrite}_x \mid \text{vwrite}_x \mid \text{action3} \) transforms to \( \text{awrite}_x \mid \text{action3} \)
- A multi-action \( \text{awrite}_x \mid \text{action3} \) transforms to \( \text{awrite}_x \mid \text{action3} \)

Similarly for actions containing a read action:

- A multi-action \( \text{aread}_x \mid \text{vread}_x \mid \text{action3} \) transforms to \( \text{aread}_x \mid \text{action3} \)
- A multi-action \( \text{aread}_x \mid \text{action3} \) transforms to \( \text{aread}_x \mid \text{action3} \)

This means that different multi-actions can transform to the same multi-action, and thus no distinction between them is possible any more. Also, an automata can now read a variable (\( \text{aread}_x \)) without consolidating (\( \text{vread}_x \)) the only process that actually ‘knows’ the value of that variable (‘VarProc_x’).

Listing A.2: Part of incorrect mCRL2 transformation (from model in Figure A.3)

```plaintext
init allow({value_x,
  a \mid \text{awrite}_x \mid \text{vread}_x,
  b \mid \text{awrite}_x \mid \text{aread}_x,
  c \mid \text{awrite}_x \mid \text{aread}_x},
comm({\text{aread}_x \mid \text{vread}_x \rightarrow \text{aread}_x},
  \text{awrite}_x \mid \text{vwrite}_x \rightarrow \text{awrite}_x),
( allow({a \mid \text{awrite}_x,
    b \mid \text{awrite}_x \mid \text{aread}_x,
    c \mid \text{awrite}_x \mid \text{aread}_x},
  comm({\text{aread}_x \mid \text{aread}_x \rightarrow \text{aread}_x},
    rename({\text{renamed}_c \rightarrow c}),
    block({c},
    comm({c \mid c \rightarrow \text{renamed}_c},
    ( \text{BehProc}_A(\text{loc}_A_{A0})
     || \text{BehProc}_B(\text{loc}_B_{B0})
     ))))))
|| \text{VarProc}_x(0)
))));
```

By rewriting Listing A.2 into Listing A.3, the previously mentioned problem is solved for the example in Figure A.3. The communication operator at line 3 now gives a unique name to a synchronised read or write action between the automata and the variable process. Also, single read or write actions without synchronisation between the automata and the variable process are now blocked by the block operator at line 1. In this way, the previously mentioned problems are resolved. With the help of A.T. Hofkamp, this solution is implemented in the mCRL2 transformation tool.
Small event-transformation bug

During the thesis, a large CIF model was transformed to mCRL2. The state space of this model in CIF was different from the state space of the generated mCRL2 model. This indicates a mistake in the transformation tool.

With the help of both the (manual) simulation tools in CIF and in mCRL2, the mistake was tracked down and a minimal working example (two automata, see Figure A.4) was designed that generates the same mistake. The incorrect part of the transformation of this CIF model to mCRL2 can be seen in Listing A.4.

The multi-action \( a \mid \text{awrite} \_x \) should be changed to an action \( a \) (without the \( \text{awrite} \_x \)). This is because there is an event \( a \) in the CIF model that does not do any additional read or writes. In Listing A.4, an \( a \) event can only happen when it also updates the variable \( y \).

Corrected mCRL2 code is given in Listing A.5. The \( a \mid \text{aread}\_x \) and \( a \mid \text{awrite}\_y \) multi-actions are unnecessary for the transformation of this specific CIF model. However, it appeared to be a lot of effort to program the transformation tool in such a way that they would not appear in the final output, that they are still there. This is no problem, as they do not change the behaviour of the system in any way.
Figure A.4: Minimal working example that results in an incorrect mCRL2 transformation.

Listing A.4: Part of an incorrect mCRL2 transformation (from the model in Figure A.4)

```mCRL2
init block({aread_y, awrite_y, vread_y, vwrite_y},
    hide({sync_y},
        comm({aread_y | vread_y -> sync_y,
                awrite_y | vwrite_y -> sync_y},
        (allow({a | awrite_y,
                a | awrite_y | aread_y,
                b | aread_y},
            (BehProc_A(loc_A_AS1)
             ||
             BehProc_B(loc_B_BS1)
             ))
            ||
            VarProc_y(false)
            )))
```

Listing A.5: Part of a corrected mCRL2 transformation (from the model in Figure A.4)

```mCRL2
init block({aread_y, awrite_y, vread_y, vwrite_y},
    hide({sync_y},
        comm({aread_y | vread_y -> sync_y,
                awrite_y | vwrite_y -> sync_y},
        (allow({a,
                a | aread_y,
                a | awrite_y,
                a | awrite_y | aread_y,
                b | aread_y},
            (BehProc_A(loc_A_AS1)
             ||
             BehProc_B(loc_B_BS1)
             ))
            ||
            VarProc_y(false)
            )))
```
Listing A.6: CIF examplar model for mCRL2 transformation purposes. Graph can be seen in Figure A.3.

```plaintext
correcting the cif to mcrl2 transformation tool

Listing A.6: CIF examplar model for mCRL2 transformation purposes. Graph can be seen in Figure A.3.

```plaintext
```

sort LocSort_A = struct loc_A_AS0 | loc_A_AS1;

proc BehProc_A(Locvar_A : LocSort_A) =
  sum x : Int . (Locvar_A == loc_A_AS0) -> a | awrite_x(0) . BehProc_A(loc_A_AS1) +
  sum x : Int . ((Locvar_A == loc_A_AS1) && (x < 5)) -> b | aread_x(x) | awrite_x(x + 1). BehProc_A(loc_A_AS1) +
  sum x : Int . ((Locvar_A == loc_A_AS1) && (x == 5)) -> c | aread_x(x) | awrite_x(0). BehProc_A(loc_A_AS0);

sort LocSort_B = struct loc_B_BS0;

proc BehProc_B(Locvar_B : LocSort_B) =
  sum x : Int . ((Locvar_B == loc_B_BS0) && (x == 5)) -> c | aread_x(x). BehProc_B(Locvar_B);

act value_x, vread_x, vwrite_x, aread_x, awrite_x : Int;

proc VarProc_x(v: Int) =
  value_x(v). VarProc_x(v) +
  vread_x(v). VarProc_x(v) +
  sum m : Int . ((m >= 0) && (m <= 6)) -> vwrite_x(m). VarProc_x(m) +
  sum m : Int . ((m >= 0) && (m <= 6)) -> vread_x(v) | vwrite_x(m). VarProc_x(m) ;

act a, renamed_a, b, renamed_b, c, renamed_c;

init allow({value_x,
  a | awrite_x | vread_x,
  b | awrite_x | aread_x,
  c | awrite_x | aread_x},
  comm({aread_x | vread_x -> aread_x,
         awrite_x | vwrite_x -> awrite_x}),
  ( allow({a | awrite_x,
            b | awrite_x | aread_x,
            c | awrite_x | aread_x},
         comm({aread_x | aread_x -> aread_x},
               rename({renamed_c -> c}),
               block({c}),
               comm({c | c -> renamed_c}),
               ( BehProc_A(loc_A_AS0) ||
                 BehProc_B(loc_B_BS0) )))||
    VarProc_x(0) )))|
Explaining some modal $\mu$-calculus notions

In this appendix, some more examples are given in order to better understand the three properties of Section 5.2. For understanding the examples in this appendix, first the syntax descriptions of Table 5.2 should be read. The examples used in this appendix are (a selection) from [11].

Comparing box and diamond modality

Clearly, $[a]\phi$ and $\langle a\rangle\phi$ express different properties. A good way to understand the differences is by giving two transition systems, one where $\langle a\rangle\phi$ holds and $[a]\phi$ is invalid, and vice versa, one where $[a]\phi$ is valid, and $\langle a\rangle\phi$ is invalid. In the first transition system of Figure B.1 $\langle a\rangle\phi$ is valid and $[a]\phi$ is not. In the second transition system the situation is reversed, namely $[a]\phi$ is valid $\langle a\rangle\phi$ is not. Both formulas are true in the third transition system and both are invalid in the fourth.

![Figure B.1: Four transition systems for explaining the differences between the diamond and box modality.](image-url)
Examples box modality

The always modality is a typical instance of a so-called safety property. Such properties typically say that something bad will never happen. A typical example is that two processes cannot be in a critical region at the same time. Entering the critical region is modelled by the action \texttt{enter} and leaving the critical region is modelled by an action \texttt{leave}. So, in a modal formula we want to say that it is impossible to do two consecutive \texttt{enter}s without a \texttt{leave} action in between:

\[ \text{true}^* \texttt{.enter} \overline{\text{.leave}}^* \texttt{.enter} \Rightarrow \text{false} \]

Another typical safety property is that there is no deadlock in any reachable state:

\[ \text{true}^* \langle \text{true} \rangle \text{true} \]

Liveness properties say that something good will eventually happen. For instance the following formula expresses that after sending a message, it can eventually be received:

\[ \text{send} \langle \text{true}^* \texttt{.receive} \rangle \text{true} \]

Compare this to the following formula

\[ \text{send} \overline{\text{.receive}}^* \langle \text{true}^* \texttt{.receive} \rangle \text{true} \]

which says that after a \texttt{send} a \texttt{receive} is possible as long as it has not happened.

Example minimal fixed point

Very often the following property is required: \( \phi \) will eventually become valid along every path. The formula to express this is:

\[ \mu X . ([\text{true}] X \lor \phi) \]

Strictly speaking, this formula will also become \texttt{true} for paths ending in a deadlock, because in such a state \( [\text{true}] X \) becomes valid. In order to avoid this anomaly, the absence of a deadlock must explicitly be mentioned:

\[ \mu X . (([\text{true}] X \land (\text{true} \langle \text{true} \rangle \text{true}) \lor \phi) \]

A variation of this is that an \texttt{a} action must unavoidably be done, provided there is no deadlock before the action \texttt{a}. 

$\mu X.[\mathbf{a} X]$ 

In order to express that $\mathbf{a}$ must be done anyhow, the possibility for a deadlock before an action $\mathbf{a}$ must explicitly be excluded. This can be expressed by the following formula:

$\mu X.(\mathbf{a} \land \langle \text{true} \rangle \text{true})$

The last two formulas are not valid for the following transition system. The reason is that the $\mathbf{b}$ can infinitely often be done, and hence, an $\mathbf{a}$ action can be avoided.

The formula $\mu X.(\mathbf{a} X \lor \langle \mathbf{a} \rangle \text{true})$ is valid in the initial state of the previous transition system. So, this transition system distinguishes between the last formula and the two before that.

For more information about fixed point modalities, the reader is referred to [11, p.74].
Hybrid CIF model UFO sorter

Main file

In the Listings below all files necessary to run the hybrid UFO sorter model are shown. First the main file (system_perbox.cif) is shown after which the remaining CIF automata are listed in alphabetical order. An elaborate explanation of the model can be found in Chapter 4.

Listing C.1: Hybrid CIF model that can detect UFO products and missing products. UFO’s can be detected at the exit conveyor. Missing products cause the conveyor to stop.

```cif
//Import of all necessary files:
svgfile \"..\abstract\usecase_conveyors.svg\";
import \"sensor_funcc.cif\";
import \"conveyor_funcc.cif\";
import \"init_empty.cif\";
```
import "generator.cif";
import "controller1.cif";
import "controller2and3.cif";
import "controller4.cif";
import "box.cif";
import "extra_contour.cif";
import "generate_delay.cif";
import "correct_event.cif";
import "counter.cif";
import "button.cif";
import "lamp.cif";
import "correct_contour.cif";
import "mystery_box.cif";
import "mystery_button.cif";
import "recognize_events.cif";

// Declaration of the four buttons to restart each conveyor after it stopped.
but1: Button(1, ctrl1.a_v, counter1.c_stop_ufo, counter1.c_stop_missing);
but2: Button(2, ctrl2.a_v, counter2.c_stop_ufo, counter2.c_stop_missing);
but3: Button(3, ctrl3.a_v, counter3.c_stop_ufo, counter3.c_stop_missing);
but4: Button(4, ctrl4.a_v, counter4.c_stop_ufo, counter4.c_stop_missing);

// Declaration of the four lamps that indicate the state of the according conveyors.
lamp1: Lamp(1, ctrl1.a_v);
lamp2: Lamp(2, ctrl2.a_v);
lamp3: Lamp(3, ctrl3.a_v);
lamp4: Lamp(4, ctrl4.a_v);
// Time
svgout id "time_txt" text value fmt("Time:%.1f", time);

// Sensor colors.
svgout id "sensor1_box" attr "fill" value if s1: "white" else "red" end;
svgout id "sensor2_box" attr "fill" value if s2: "white" else "red" end;
svgout id "sensor3_box" attr "fill" value if s3: "white" else "red" end;
svgout id "sensor4_box" attr "fill" value if s4: "white" else "red" end;

// The generator.
gen: GeneratorStochastic(10.0, 12.0, ctrl1.dieback, ctrl1.stop);

// The distribution for determining the delays given to each box to create error behavior.
    // Was -2.0 , 2.0
Behaviour: Delay( gen.c_p0, gen.c_p1, gen.c_p2, -0.01 , 0.01);

// The mysterybox and mysterybutton declaration.
mysterybox: MysteryBox( 10, 10.0, ctrl1.a_v, ctrl2.a_v, ctrl3.a_v, ctrl4.a_v, mysterybutton.u_a_push);
mysterybutton: MysteryButton(10);

// The declaration of the four sensors for each conveyor.
alg bool s1 = BoxesAtSensor(box1.x, box2.x, box3.x, mysterybox.x, 5.0, 9.0);
alg bool s2 = BoxesAtSensor(box1.x, box2.x, box3.x, mysterybox.x, 11.0, 15.0);
alg bool s3 = BoxesAtSensor(box1.x, box2.x, box3.x, mysterybox.x, 17.0, 21.0);
alg bool s4 = BoxesAtSensor(box1.x, box2.x, box3.x, mysterybox.x, 23.0, 27.0);

// Declaration of each box
box1: Box(1, xinit1, Behaviour.delay, ctrl1.a_v, ctrl2.a_v, ctrl3.a_v, ctrl4.a_v, gen.c_p0);
box2: Box(2, xinit2, Behaviour.delay, ctrl1.a_v, ctrl2.a_v, ctrl3.a_v, ctrl4.a_v, gen.c_p1);
box3: Box(3, xinit3, Behaviour.delay, ctrl1.a_v, ctrl2.a_v, ctrl3.a_v, ctrl4.a_v, gen.c_p2);

// Declaration of each frame corresponding to a nominal position of a box send by the generator
nom1: Nominal(1, xinit1, 5.0, 11.0, 17.0, 23.0, ctrl1.a_v, ctrl2.a_v, ctrl3.a_v, ctrl4.a_v, gen.c_p0,
            events_at_1.c_correct_1, events_at_2.c_correct_1, events_at_3.c_correct_1, events_at_4.c_correct_1,
            correct1.c_delete_1, correct2.c_delete_1, correct3.c_delete_1, correct4.c_delete_1);
nom2: Nominal(2, xinit2, 5.0, 11.0, 17.0, 23.0, ctrl1.a_v, ctrl2.a_v, ctrl3.a_v, ctrl4.a_v, gen.c_p1,
            events_at_1.c_correct_2, events_at_2.c_correct_2, events_at_3.c_correct_2, events_at_4.c_correct_2,
nom3: Nominal(3, xinit3, 5.0, 11.0, 17.0, 23.0, ctrl1.a_v, ctrl2.a_v, ctrl3.a_v, ctrl4.a_v, gen.c_p2,
    events_at_1.c_correct_3, events_at_2.c_correct_3, events_at_3.c_correct_3, events_at_4.c_correct_3,
    correct1.c_delete_3, correct2.c_delete_3, correct3.c_delete_3, correct4.c_delete_3);

// Declaration of the three contours used to correct UFO events.
nom4_extra: Nominal_extra(4, xinit1, 5.0, 11.0, 17.0, 23.0, ctrl1.a_v, ctrl2.a_v, ctrl3.a_v, ctrl4.a_v,
    correct1.c_extra_1, correct2.c_extra_1, correct3.c_extra_1, correct4.c_extra_1);

nom5_extra: Nominal_extra(5, xinit2, 5.0, 11.0, 17.0, 23.0, ctrl1.a_v, ctrl2.a_v, ctrl3.a_v, ctrl4.a_v,
    correct1.c_extra_2, correct2.c_extra_2, correct3.c_extra_2, correct4.c_extra_2);

nom6_extra: Nominal_extra(6, xinit3, 5.0, 11.0, 17.0, 23.0, ctrl1.a_v, ctrl2.a_v, ctrl3.a_v, ctrl4.a_v,
    correct1.c_extra_3, correct2.c_extra_3, correct3.c_extra_3, correct4.c_extra_3);

// Declaration of the four controllers, one for each conveyor. The controller of conveyor 2 and 3 are
// equivalent.
ctrl1: Control(1, s1, conveyor_stop234(ctrl2.stop, ctrl3.stop), ctrl2.leaving, but1.pushed,
    gen.c_pc, counter1.c_stop_ufo, counter1.c_stop_missing );

ctrl12: Control23(2, s2, s1, conveyor_stop34(ctrl3.stop, ctrl4.stop), ctrl3.leaving, but2.pushed, but1.pushed,
    ctrl1.c_pcout, counter2.c_stop_ufo, counter2.c_stop_missing );

ctrl13: Control23(3, s3, s2, conveyor_stop4(ctrl4.stop), ctrl4.leaving, but3.pushed, but2.pushed, ctrl2.c_pcout,
    counter3.c_stop_ufo, counter3.c_stop_missing );

ctrl14: Control4(4, s4, s3, but4.pushed, but3.pushed, ctrl3.c_pcout, counter4.c_stop_ufo,
counter4.c_stop_missing );

// Declaration of the correct automaton, which correct events at each conveyor.
correct1: Correct(events_at_1.c_ufo, events_at_1.c_miss, 0.0, 24.0, nom1.x, nom2.x, nom3.x, nom4_extra.x,
                    nom5_extra.x, nom6_extra.x);
correct2: Correct(events_at_2.c_ufo, events_at_2.c_miss, 0.0, 24.0, nom1.x, nom2.x, nom3.x, nom4_extra.x,
                    nom5_extra.x, nom6_extra.x);
correct3: Correct(events_at_3.c_ufo, events_at_3.c_miss, 0.0, 24.0, nom1.x, nom2.x, nom3.x, nom4_extra.x,
                    nom5_extra.x, nom6_extra.x);
correct4: Correct(events_at_4.c_ufo, events_at_4.c_miss, 0.0, 24.0, nom1.x, nom2.x, nom3.x, nom4_extra.x,
                    nom5_extra.x, nom6_extra.x);

// Declaration of the automaton that recognizes events at each conveyor.
events_at_1: UFO_Missing(s1, ( nom1.x, nom2.x, nom3.x, nom4_extra.x, nom5_extra.x, nom6_extra.x ), 4.5, 5.0,
                  5.5, correct1.c_corrected);
events_at_2: UFO_Missing(s2, ( nom1.x, nom2.x, nom3.x, nom4_extra.x, nom5_extra.x, nom6_extra.x ), 10.5, 11.0,
                  11.5, correct2.c_corrected);
events_at_3: UFO_Missing(s3, ( nom1.x, nom2.x, nom3.x, nom4_extra.x, nom5_extra.x, nom6_extra.x ), 16.5, 17.0,
                  17.5, correct3.c_corrected);
events_at_4: UFO_Missing(s4, ( nom1.x, nom2.x, nom3.x, nom4_extra.x, nom5_extra.x, nom6_extra.x ), 22.5, 23.0,
                  23.5, correct4.c_corrected);

// Declaration of the counters that keep track of the events and gives a signal if an error occurs.
counter1: Counter(1, but1.pushed, events_at_1.c_ufo, events_at_1.c_miss);
counter2: Counter(2, but2.pushed, events_at_2.c_ufo, events_at_2.c_miss);
counter3: Counter(3, but3.pushed, events_at_3.c_ufo, events_at_3.c_miss);
counter4: Counter(4, but4.pushed, events_at_4.c_ufo, events_at_4.c_miss);

// Conveyer colors
svgout id "belt1_outer" attr "fill"
value switch ctrl1.a_v:
case 0 : "rgb(220,220,220)"
case 1 : "rgb(33,33,33)"
else "rgb(33,33,33)"
end;

svgout id "belt2_outer" attr "fill"
value switch ctrl2.a_v:
case 0 : "rgb(220,220,220)"
case 1 : "rgb(33,33,33)"
else "rgb(33,33,33)"
end;

svgout id "belt3_outer" attr "fill"
value switch ctrl3.a_v:
case 0 : "rgb(220,220,220)"
case 1 : "rgb(33,33,33)"
else "rgb(33,33,33)"
end;

svgout id "belt4_outer" attr "fill"
value switch ctrl4.a_v:
case 0 : "rgb(220,220,220)"
case 1 : "rgb(33,33,33)"
else "rgb(33,33,33)"
end;

svgout id "product_type" text value if
(box1.x >= 20 and box1.UFO) or
(box2.x >= 20 and box2.UFO) or
(box3.x >= 20 and box3.UFO) or
(mysterybox.x >= 20 and mysterybox.UFO): "UFO" else "No UFO" end;
Declaration of the boxes

**Listing C.2:** The hybrid automaton definition *Box.*

```plaintext
import "system_perbox.cif";

// This automaton Box represents the boxes of the system, it contains the int nr to distinguish the boxes from eachother. xinit is the initial position of the box (-1), delay is the prescribed delay which is used to create the desired errors. a_v are the different velocities of each conveyor and c_pin is the event on which it corresponds with the generator at the moment a new box is generated.

automaton def Box(alg int nr; alg real xinit, delay; alg int a_v1, a_v2, a_v3, a_v4; controllable c_pin):

    event goToWaiting, goToFirstCnvr, goToStart;
    cont x = xinit;
    disc bool UFO = false;
    disc bool Missing = false;

    // This location initializes each position x of a box
    location Initialize:
        equation x' = 0.0;
        initial;
        edge goToWaiting when x = -1.0 goto Waiting;
        edge goToFirstCnvr when x >= 0.0 goto Moving;

    // edge when the generator gives a signal to send a box this location correct the starting position by a delay
    // this is used to construct behavior that contains UFO and Missing errors
    location Waiting:
        equation x' = 0.0;
        edge c_pin do x := 0.0 - delay goto Moving;

    // The velocity of a box is a function of the position and the corresponding
    // velocity of the current conveyor
    location Moving:
        equation x' =
            if x < 8
            elseif x >= 8 and x < 14
            elseif x >= 14 and x < 20
            elseif x >= 20 and x < 26
            else
            end;
```

\begin{verbatim}
edge when x < 8  and counter1.UFO and not UFO and not Missing do UFO := true;
edge when x >= 8 and x < 14 and counter2.UFO and not UFO and not Missing do UFO := true;
edge when x >= 14 and x < 20 and counter3.UFO and not UFO and not Missing do UFO := true;
edge when x >= 20 and x < 26 and counter4.UFO and not UFO and not Missing do UFO := true;

edge when x < 8  and counter1.Missing and not UFO and not Missing do Missing := true;
edge when x >= 8 and x < 14 and counter2.Missing and not UFO and not Missing do Missing := true;
edge when x >= 14 and x < 20 and counter3.Missing and not UFO and not Missing do Missing := true;
edge when x >= 20 and x < 26 and counter4.Missing and not UFO and not Missing do Missing := true;

edge goToStart when x >= 28 do UFO := false, Missing := false, x := -1.0 goto Waiting;

// Plotting the boxes at the correct position
svgout id "box" + <string>nr attr "x" value 30 * x + 10.5;

// Change color of box if it is an UFO box
svgout id "box" + <string>nr attr "fill" value if UFO: "red" else "green" end;

// Allow the box only to be visible if it is on one of the conveyors
svgout id "box" + <string>nr attr "visibility" value if x > -0.5: "visible" else "hidden" end;

end
\end{verbatim}
Button in case of an error

Listing C.3: The hybrid automaton definition *Button*.

```cif
// This is a standard automaton to control the button that needs to be pushed to restart a conveyor
// at which an error has occurred. The int nr is to distinguish the different buttons from each other, the
// events c_stop_ufo and c_stop_missing corresponds to the events from the automaton Counter in the
// counter.cif file. They occur when one of the two errors appears.

automaton def Button(alg int nr, a_v; controllable c_stop_ufo, c_stop_missing):

 uncontrollable u_a_push; // the uncontrollable event to push the button
 cont t_reset der 1.0;

 event autoPush, resetButton, resetClock;

disc bool pushed = false; // the boolean "pushed" is used to track if the button is pushed or not.

t // In this location the button can be pushed to undo the error.
location PushMissing:
  edge u_a_push when t_reset > 1 do pushed := true, t_reset := 0 goto Wait;

// When a UFO occurs, the button will be pushed automatically after 1 second.
location autoPushUFO:
  edge autoPush when t_reset > 1 do pushed := true, t_reset := 0.0 goto Wait;
  edge u_a_push;

// If there is no error, this automaton is in this location, it waits until one of the two error events
// occur and if so, it goes again to the location Push. When it comes from the location Push into this
// location, the boolean pushed will be set to false after 2 seconds. This is needed in order for the
// following conveyor to go to the Running state, so that a corrected error will cause that the box can
// start running immediately.
location Wait:
  initial;
  edge resetButton when pushed and t_reset > 2.0 do pushed := false;
  edge resetClock when pushed and t_reset > 1.0 and a_v < 1 do t_reset := 0.0;
  edge c_stop_ufo when not pushed do t_reset := 0.0 goto autoPushUFO;
  edge c_stop_missing when not pushed goto PushMissing;
  edge c_stop_ufo when pushed;
  edge c_stop_missing when pushed;
  edge u_a_push;
```

// This is a standard automaton to control the button that needs to be pushed to restart a conveyor
// at which an error has occurred. The int nr is to distinguish the different buttons from each other, the
// events c_stop_ufo and c_stop_missing corresponds to the events from the automaton Counter in the
// counter.cif file. They occur when one of the two errors appears.
// Here the input of pushing the button is created.
svg in id "button" + <string>nr event u_a_push;

// This svg element colors the button orange at the moment it can be pushed.
svg out id "button" + <string>nr attr "fill" value if PushMissing: "orange" else "black" end;

end
Controller first conveyor

Listing C.4: The hybrid automaton definition Control.

```
// The three controllers in this system have many aspects in common. The only differences between them are
// that controller 1 and 4 does not need that much information compared to the controller of conveyor 2 and 3.
// This means these two controllers has less input arguments and that is the main reason why three different
// automatons are made.

automaton def Control(alg int nr; alg bool s, stop_else, leaving_next, pushed;
                        controllable c_pcin, c_stop_ufo, c_stop_missing):
                        controllable c_pcout;
                        disc int a_v = 1;
                        cont t = 0.0 der 1.0;
                        event pLeft, pArrivedNxtCnvr, stopP, cnvrDieback, goToRunning, errorCleared;

                        // The variables leaving and entering represent the state of the conveyor, if a box is entering, leaving or
                        // both.
                        // The variable left represents the fact if a box has left the current conveyor or not. This value is true
                        // when the sensor of the current conveyor turns off and left is false if a box, probably the same box, is
                        // leaving at the succeeding conveyor. This variable is used to model the fact that boxes cannot follow
                        // each other too fast.
                        disc bool leaving = false; // determines whether a box is leaving or not
                        disc bool entering = false; // determines whether a box is entering or not
                        disc bool left = false; // determines whether the last box has left and/or arrived at the succeeding
                        // conveyor.

                        // The variables stop and dieback represents the fact if the current conveyor is in stop state or in dieback
                        // state. These states speaks for them self.
                        disc bool stop = false; // determines whether the conveyor has stopped
                        disc bool dieback = false; // determines whether the conveyor has diebacked
```
// In this first location the system is initializing. During this process a box can appear which results in
// a transition to the state running. Next to this it can be needed to stop the conveyor if an error occurred
// at the conveyor or a succeeding error is in stop state (stop_else = true) and this conveyor needs to
dieback. When non of the above described situations happen, the controller will go to the Running
// state after 3.0 seconds.

location Initialize:
initial;
edge when s do t := 0.0 goto Running;
edge when t >= 3.0 do t := 0.0 goto Running;
edge when stop do a_v := 0 goto Stopped;
edge when stop_else do a_v := 0 goto DieBack;

// This state is the most complicated one, it represents the conveyor when it is running. During this mode
// different situation can appear, so each line is shortly explained below.

location Running:

// An box can be send from the generator in the cases when a box is leaving or nothing is happening. In
// both cases entering will become true and the time is set to zero.
edge c_pcin when not entering and not leaving do t := 0.0, a_v := 1, entering := true;
edge c_pcin when leaving and not entering do t := 0.0, a_v := 1, entering := true;

// When the sensor is off and the conveyor is in leaving state, meaning a box has left, the variable left
// becomes true and leaving becomes false. The box has left.
edge pLeft when not s and leaving do leaving := false, left := true, t := 0.0;

// When a box has left the current conveyor (left = true) and a box is leaving at the next conveyor
// (leaving_next) this means the box is definitely arrived, so the value of left is false again.
edge pArrivedNxtCnvr when leaving_next and left do left := false;

// When the sensor at the current conveyor is on, a box is left but not arrived at the next conveyors
// sensor (left = true). At the same time there is no box leaving or an UFO or Missing event occurs, than
// this is the moment to stop the conveyor because otherwise boxes will follow each other too fast.
edge stopP when s and left and not leaving do a_v := 0, stop := true goto Stopped;

// When the conveyor is running for 8.0 time units without the sensor going on or an box entering it
// turns off, this to save energy.
edge when t >= 8.0 do a_v := 0 goto Stopped;

// When a error occurs at the current conveyor these synchronizing events make sure the conveyor will
// stop and go to one of the two error states.
edge c_stop_ufo do a_v := 0, stop := true goto UFO_error;
edge c_stop_missing do a_v := 0, stop := true goto Missing_error;

// When a succeeding conveyor has stopped (stop_else = true) the current conveyor needs to dieback.
edge cnvrDieback when stop_else do a_v := 0, dieback := true goto DieBack;

// When the sensor at the current conveyor is on, no box is leaving and no box is arriving at the next
// conveyor than the box at the current conveyor can be send to the next conveyor.
edge c_pcout when s and not leaving and not left do t := 0.0, a_v := 1, leaving := true,
  entering := false;

// This location represents the situation in which a conveyor has stopped.
location Stopped:
  // When the conveyor stopped because of time restriction it can restart again when a new box is entering.
  edge c_pcin when not leaving do a_v := 1, t := 0.0, leaving := false,
    entering := true, stop := false goto Running;
  // When an error occurs at the current conveyor and this conveyor is also stopped due to the intermediate
  // distance the error gets preference. These synchronizing events make sure the conveyor will
  // stop and go to one of the two error states.
  edge c_stop_ufo goto UFO_error;
  edge c_stop_missing goto Missing_error;
  // When the sensor is on and left is false probably a box that left has arrived at the next conveyor and
  // the current conveyor can start again.
  edge goToRunning when s and not left do a_v := 1, t := 0.0, stop := false goto Running;
  // When a box is leaving at the next conveyor it has arrived at that conveyor so left can be false again.
  edge pArrivedNxtCnvr when leaving_next and left do left := false;

// This location represents the behavior in case of an UFO error.
location UFO_error:
  // When during the UFO error another box arrives at the next conveyor the left value is updated.
  edge pArrivedNxtCnvr when leaving_next and left do left := false;
  // When the button corresponding with this conveyor is pushed (pushed = true) the conveyor can start again.
  // This because it is assumed that the error is cleared.
  edge errorCleared when pushed and not left do stop := false, a_v := 1 goto Running;
// This location represents the behavior in case of an Missing error.
location Missing_error:

// Here also the left value is updated when necessary during the error state.
edge pArrivedNxtCnvr when leaving_next and left do left := false;

// When the button corresponding with this conveyor is pushed (pushed = true) the conveyor can start again.
// This because it is assumed the error is cleared.
edge errorCleared when pushed do a_v := 1, t := 0.0, stop := false goto Running;

// The dieback situation
location DieBack:

// When non of the succeeding conveyors is stopped (not stop_else) the conveyor can start again.
edge goToRunning when not stop_else do a_v := 1, t := 0.0, dieback := false goto Running;

// Controller state
svgout id "ctrl" + <string>nr + "," + "_state" text value <string> self;
end
Controller for middle conveyors

Listing C.5: The hybrid automaton definition Control23.

// This controller is almost the same as controller 1, so here only the differences will be explained.

```cif
automaton def Control23(alg int nr; alg bool s, s_prev, stop_else, leaving_next, pushed,
   pushed_prev; controllable c_pcin, c_stop_ufo, c_stop_missing ):

  controllable c_pcout;
  disc int a_v = 1;
  cont t = 0.0 der 1.0;
  event pLeft, pArrivedNxtCnvr, pEntering, stopP, cnvrDieback, goToRunning, errorCleared, prevButtonPushed;

disc bool leaving = false;
disc bool entering = false;
disc bool left = false;
disc bool stop = false;
disc bool dieback = false;

location Initialize:
  initial;
  edge when s do t := 0.0 goto Running;
  edge when t >= 3.0 do t := 0.0 goto Running;
  edge when stop do a_v := 0 goto Stopped;
  edge when stop_else do a_v := 0 goto DieBack;

location Running:
  edge c_pcin when not entering and not leaving do t := 0.0, a_v := 1, entering := true;
  edge c_pcin when leaving and not entering do t := 0.0, a_v := 1, entering := true;
  edge pLeft when not s and leaving do leaving := false, left := true, t := 0.0;
  edge pArrivedNxtCnvr when leaving_next and left do left := false;
  edge stopP when s and left and not leaving do a_v := 0, stop := true goto Stopped;

// When the sensor of the previous conveyor is off (not s_prev) and the current sensor is on and entering
// is still true it becomes false because a box is leaving and no box is entering.
```
edge when not s_prev and s and entering do entering := false;
edge when t >= 8.0 do a_v := 0 goto Stopped;
edge c_stop_ufo do a_v := 0, stop := true goto UFO_error;
edge c_stop_missing do a_v := 0, stop := true goto Missing_error;
edge cnvrDieback when stop_else do a_v := 0, dieback := true goto DieBack;

// When the button from the preceding conveyor is pushed, the current conveyor will start, and to be
// sure this will not happen again and again the restriction of 2.0 time units is given.
edge prevButtonPushed when pushed_prev and t > 2.0 do a_v := 1, t := 0.0;
edge c_pcout when s and not leaving and not left do t := 0.0, a_v := 1, leaving := true,
entering := false;

location Stopped:
edge c_pcin when not leaving do a_v := 1, t := 0.0, leaving := false,
entering := true, stop := false goto Running;

// When an error occurs at the current conveyor and this conveyor is also stopped due to the intermediate
// distance the error gets preference. These synchronizing events make sure the conveyor will
// stop and go to one of the two error states.
edge c_stop_ufo goto UFO_error;
edge c_stop_missing goto Missing_error;

// When the button of the preceding conveyor is pushed it means the error at that conveyor is repaired, so
// a box is coming.
edge prevButtonPushed when pushed-prev do a_v := 1, t := 0.0, stop := false goto Running;
edge goToRunning when s and not left do a_v := 1, t := 0.0, stop := false goto Running;
edge pArrivedNxtCnvr when leaving_next and left do left := false;

location UFO_error:
edge pArrivedNxtCnvr when leaving_next and left do left := false;

// When the sensor of the previous conveyor is on and no box is entering, entering becomes true.
edge pEntering when s_prev and not entering do entering := true;

edge errorCleared when pushed do a_v := 1, stop := false goto Running;

location Missing_error:

edge pArrivedNxtCnvr when leaving_next and left do left := false;

edge errorCleared when pushed do a_v := 1, t := 0.0, stop := false goto Running;

location DieBack:

edge goToRunning when not stop_else do a_v := 1, t := 0.0, dieback := false goto Running;

// Controller state
svgout id "ctrl" + <string>nr + "_state" text value <string>self;

end
Controller for last conveyors

Listing C.6: The hybrid automaton definition \textit{Control4}.

// This controller is almost the same as controller 1, only differences are that the values of die back, left // and leaving\_next are not in this controller. This because it is the last conveyor of the line, so in this // assignment it can not dieback and does not need to take into account the intermediate distance between // boxes.

\begin{verbatim}
automaton def Control4(alg int nr; alg bool s, s\_prev, pushed, pushed\_prev; controllable c\_pcin, c\_stop\_ufo, c\_stop\_missing):
  controllable c\_pcout;
  disc int a\_v = 1;
  cont t = 0.0 der 1.0;
  event p\_Left, p\_Entering, go\_To\_Running, error\_Cleared, prev\_Button\_Pushed;

  disc bool leaving = false;
  disc bool entering = false;
  disc bool stop = false;

  location Initialize:
    initial;
    edge when s do t := 0.0 goto Running;
    edge when t >= 3.0 do t := 0.0 goto Running;
    edge when stop do a\_v := 0 goto Stopped;

  location Running:
    edge c\_pcin when not entering and not leaving do t := 0.0, a\_v := 1, entering := true;
    edge c\_pcin when leaving and not entering do t := 0.0, a\_v := 1, entering := true;
    edge p\_Left when not s and leaving do leaving := false;
    edge when t >= 8.0 do a\_v := 0 goto Stopped;
    edge c\_stop\_ufo do a\_v := 0, stop := true goto UFO\_error;
    edge c\_stop\_missing do a\_v := 0, stop := true goto Missing\_error;
    edge prev\_Button\_Pushed when pushed\_prev and t > 2.0 do a\_v := 1, t := 0.0;
    edge c\_pcout when s and entering and not leaving do t := 0.0, a\_v := 1, leaving := true,

\end{verbatim}
entering := false;

location Stopped:
  edge c_pcin when not leaving do a_v := 1, t := 0.0, leaving := false,
  entering := true, stop := false goto Running;
  edge goToRunning when s do a_v := 1, t := 0.0, leaving := true goto Running;

location UFO_error:
  edge pEntering when s_prev and not entering do entering := true;
  edge errorCleared when pushed do a_v := 1, t := 0.0, stop := false goto Running;

location Missing_error:
  edge errorCleared when pushed do a_v := 1, t := 0.0, stop := false goto Running;

// Controller state
svgout id "ctrl" + <string>nr + "_state" text value <string>self;
end
Functions used for the conveyor

Listing C.7: The functions used by the conveyors.

// These functions are used to determine whether succeeding conveyors are stopped or not, because if so the
// current conveyor needs to dieback.

func bool conveyor_stop234(bool x1, x2, x3):
    return x1 or x2 or x3;
end

func bool conveyor_stop34(bool x1, x2):
    return x1 or x2;
end

func bool conveyor_stop4(bool x1):
    return x1;
end
Updating of nominal positions

Listing C.8: The hybrid automaton Nominal.

// This automaton Nominal is used to correct events because the box and its contour are both in the interval of 0.5. This automaton also deletes contours that cause a missing event at the conveyors. By deleting these contours the missing event will not occur at the sensors at the following conveyors.

automaton def Nominal(alg int nr; alg real xinit, pos1, pos2, pos3, pos4; alg int a_v1, a_v2, a_v3, a_v4; controllable c_pin, c_corr1, c_corr2, c_corr3, c_corr4, c_del1, c_del2, c_del3, c_del4):

cont x = xinit;

event goToWaiting, goToFirstCnvr, goToStart;

location Initialize:
  equation x' = 0.0;
  initial;
  edge goToWaiting when x = -1.0 goto Waiting;
  edge goToFirstCnvr when x >= 0.0 goto Moving;

location Waiting:
  equation x' = 0.0;
  edge c_pin do x := 0.0 goto Moving;

location Moving:
  equation x' =
    if x < 8 : a_v1 * 1.0
    elif x >= 8 and x < 14 : a_v2 * 1.0
    elif x >= 14 and x < 20 : a_v3 * 1.0
    elif x >= 20 and x < 26 : a_v4 * 1.0
    else
      1.0
  end;
  edge goToStart when x >= 28 do x := -1.0 goto Waiting;

// By these events the contour positions are corrected to the sensor positions, this because the moment the frontside of the box is at a sensor can be determined exactly.

edge c_corr1 do x := pos1 goto Corrected;
edge c_corr2 do x := pos2 goto Corrected;
edge c_corr3 do x := pos3 goto Corrected;
edge c_corr4 do x := pos4 goto Corrected;
With these events contours that cause a missing event can be deleted, they get the position -1.0 again.

```plaintext
edge c_del1 do x := -1.0 goto Waiting;
edge c_del2 do x := -1.0 goto Waiting;
edge c_del3 do x := -1.0 goto Waiting;
edge c_del4 do x := -1.0 goto Waiting;

location Corrected:

equation x' =
if x < 8 : a_v1 * 1.0
elif x >= 8 and x < 14 : a_v2 * 1.0
elif x >= 14 and x < 20 : a_v3 * 1.0
elif x >= 20 and x < 26 : a_v4 * 1.0
else
    1.0
end;

edge goToStart when x >= 28 do x := -1.0 goto Waiting;

// Box position nominal
svgout id "box" + <string>nr + "_nom" attr "x" value 30 * (x) + 10.5;

// Box visibility nominal
svgout id "box" + <string>nr + "_nom" attr "visibility" value if (x) > -0.5: "visible" else "hidden" end;
end
```
Correcting of missing or UFO events

Listing C.9: The hybrid automaton `Correct`.

```cif
import "sensor_funcs.cif";

// This automaton gives the signal to correct the current event that appears in the system.
// UFO events are corrected by placing a new frame around the box, Missing events are corrected by
// deleting the current frame. By the synchronizing events c_ufo and c_miss the event that appears is known.
// Then in the current state the events c_extra or c_delete gives the signal to correct the event. c_extra
// makes sure a new frame is placed around a box with UFO error. c_delete gives the signal to delete a frame
// for which no box occurs.

automaton def Correct(controllable c_ufo, c_miss; alg real pos1, pos2;

    alg real nom1, nom2, nom3, nom4, nom5, nom6):

    controllable c_corrected, c_extra_1, c_extra_2, c_extra_3, c_delete_1, c_delete_2, c_delete_3;

    // By these three boolean variables it is defined if an extra frame (nom4, nom5 and nom6) is at the current
    // conveyor, because there is a automaton correct for each conveyor.
    // This because if it is not at the conveyors it can be used to correct an UFO error.
    alg bool extra1 = xAtSensor(nom4, pos1, pos2);
    alg bool extra2 = xAtSensor(nom5, pos1, pos2);
    alg bool extra3 = xAtSensor(nom6, pos1, pos2);

    // By these three boolean variables it is defined which of the three actual frames (nom1, nom2 and nom3) is
    // at the current conveyor, because there is an automaton correct for each conveyor. If this is true the
    // system knows which frame to delete because of a missing error.
    alg bool nom_1 = xAtSensor(nom1, pos1, pos2);
    alg bool nom_2 = xAtSensor(nom2, pos1, pos2);
    alg bool nom_3 = xAtSensor(nom3, pos1, pos2);

    // This is the initial location of the automaton, changing of location will appear after an missing
    // or ufo event has occured.
    location start:
        initial;
        edge c_ufo goto UFO_event;
        edge c_miss goto Missing_event;

        // In this location an UFO error has occurred and the signal to correct this is given after it is determined
```

// The function from the sensor_funcs.cif file are imported because they are used in this automaton.
// which extra frame is free.

location UFO_event:

edge c_extra_1 when extra1 = false goto Corrected;
edge c_extra_2 when extra2 = false goto Corrected;
edge c_extra_3 when extra3 = false goto Corrected;

// In this location a missing error has occurred and the signal to correct this is given after it is
// determined which frame is at the conveyor without a box corresponding to it.

location Missing_event:

edge c_delete_1 when nom_1 = true goto Corrected;
edge c_delete_2 when nom_2 = true goto Corrected;
edge c_delete_3 when nom_3 = true goto Corrected;

// After the correcting event has taken place the automaton comes in this Corrected location at which the
// synchronizing event c_corrected takes place, this gives the signal to the Error automaton that the error
// is corrected.

location Corrected:

edge c_corrected goto start;

end
Listing C.10: The hybrid automaton `Counter`.

```plaintext
automaton def Counter(alg int nr; alg bool pushed; controllable c_ufo, c_miss):
    controllable c_stop_ufo, c_stop_missing;

disc tuple( string a; string b; string c ) errors = ("","","");

disc bool UFO = false;

disc bool Missing = false;

event errorCleared;

// This is the initial location in which the current event UFO or Missing is saved in the tuple and depending
// on the error the automaton will be in the location of that current event.
location Clean: initial;

edge c_ufo do errors := ( errors[1], errors[2], "UFO" ), UFO := true goto UFOError;

edge c_miss do errors := ( errors[1], errors[2], "Missing" ), Missing := true goto MissingError;

// When an UFO error is recognized the automaton will be in this state, the synchronizing event c_stop_ufo
// will stop the conveyor. Then the variable UFO becomes false and the whole process starts over again so
// the automaton will return to the state Clean.
location UFOError:

    edge c_stop_ufo goto UFOErrorReset;

location UFOErrorReset:

    edge errorCleared when pushed do UFO := false goto Clean;

// When an Missing error is recognized the automaton will be in this state, the synchronizing event
// c_stop_missing will stop the conveyor. Then the variable missing becomes false and the whole process
// starts over again so the automaton will return to the state Clean.
location MissingError:
```
edge c_stop_missing

location MissingErrorReset:
  edge errorCleared when pushed do Missing := false
  goto Clean;

// The svg element to plot the string of the first element in the tuple
svgout id "error1" + <string>nr text value fmt("1.\%s", errors[0]);

// The svg element to plot the string of the second element in the tuple
svgout id "error2" + <string>nr text value fmt("2.\%s", errors[1]);

// The svg element to plot the string of the third element in the tuple
svgout id "error3" + <string>nr text value fmt("3.\%s", errors[2]);
end
Creating new contours

Listing C.11: The hybrid automaton *Nominal_extra*.

// This automaton Nominal_extra creates the extra frames (4, 5, 6) which are placed around a box that creates an UFO event. This is done such that next conveyors will not recognize this UFO event again. By doing this the event is corrected.

```plaintext
automaton def Nominal_extra(alg int nr; alg real xinit, pos1, pos2, pos3, pos4; alg int a_v1, a_v2, a_v3, a_v4;
controllable c_pin1, c_pin2, c_pin3, c_pin4):

cont x = xinit;

location Initialize:
equation x' = 0.0;
initial;
edge when x = -1.0 goto Waiting;
edge when x >= 0.0 goto Moving;

// In this location the starting position of the frame is attached to the variable x. These are the positions of the sensors, because we can determine the moment a box is in the sensor exactly. A shortcoming of this is that it is not possible to drop boxes under a sensor because then the frame is placed wrong. But since it is assumed that boxes can not be put under the sensor this will work perfectly.

location Waiting:
equation x' = 0.0;
edge c_pin1 do x:= pos1 goto Moving;
edge c_pin2 do x:= pos2 goto Moving;
edge c_pin3 do x:= pos3 goto Moving;
edge c_pin4 do x:= pos4 goto Moving;

location Moving:
equation x' = if x < 8 : a_v1 * 1.0
elif x >= 8 and x < 14 : a_v2 * 1.0
elif x >= 14 and x < 20 : a_v3 * 1.0
elif x >= 20 and x < 26 : a_v4 * 1.0
else 1.0
end;
edge when x >= 28 do x := -1.0 goto Waiting;

// Box position nominal
svgout id "box" + <string>nr + "_nom" attr "x" value 30 * (x) + 10.5;
```
// Box visibility nominal
svgout id "box" + <string>nr + "_nom" attr "visibility" value if (x) > -0.5: "visible" else "hidden" end;
end
A Tooldef file

Listing C.12: A tooldef file for simulating the model using the svg visualization.

```python
from "lib:cif3" import *

cif3sim(
    "system_perbox.cif",
    "i_svg",
    "--stateviz=true",
    "--distributions-seed=3",
    "--solver-root-maxchk=0.1",
    "--frame-rate=auto",
    "--speed=1.0",
    "--option-dialog=no",
)
```

Generating the delays of the boxes

Listing C.13: The hybrid automaton Delay.

// This automaton Delay is a simple one which defines the delay that are given to the boxes after the generator
// gives a start signal. These can be defined by the user or chosen random from a distribution.
automaton def Delay(controllable c_generate_1, c_generate_2, c_generate_3; alg real d_next_min, d_next_max):

// The uniform distribution from which a delay is chosen, it depends on the min and max value which can
// be said on the top level of the code.
disc dist real d = uniform(d_next_min, d_next_max);
disc real delay;

// In this location the automaton reacts on a generating event from the generator after which the delay
// is determined and shared with the box automaton.
location DetermineDelay:
    initial;
    edge c_generate_1 do (delay, d) := sample d;
    edge c_generate_2 do (delay, d) := sample d;
    edge c_generate_3 do (delay, d) := sample d;
end
Generating the boxes

Listing C.14: The hybrid automaton \textit{GeneratorStochastic}.

// This automaton describes the generator that generate boxes at the beginning of the belt. This automaton
// is for a big part based on the stochastic generator in the lecture notes, and generates boxes with in a
// stochastic behaviour.

\begin{verbatim}
automaton def GeneratorStochastic(alg real t_next_min, t_next_max; alg bool dieback, stop):
    controllable c_p0, c_p1, c_p2;
    controllable c_pc;
    cont t_next der -1.0; // timer box inter arrival time
    cont t_atend der -1.0; // timer box at conveyor edge
    disc dist real d = uniform(t_next_min, t_next_max);
    disc real clock = 0.0;

    // In this location the generator waits until the time that it is able to send a new box. By means of the
    // variable clock the time of the t_next at which conveyor 1 is in dieback or stop state is saved so that in
    // case conveyor 1 is restarted the generator waits the remaining time, this to ensure no boxes will be
    // generated on top of each other.
    location WaitForNextBox:
        edge when (dieback or stop) and clock = 0.0 do clock := t_next;
        edge when t_next <= 0 and clock = 0.0 goto SignalController;
        edge when not (dieback or stop) and clock > 0.0 do t_next := clock, clock := 0.0;

    // The SignalController sends a signal to the controller of conveyor 1 that a box is coming. At the same
    // time it generates the time to wait for the next box (t_next).
    location SignalController:
        initial;
        edge c_pc when t_next <= 0 do t_atend := 1.0 goto SendBox;

    // In this location the box is actually send to conveyor 1. This is done after 1.0 time units because the
    // starting position of each box is -1.0.
    location SendBox:
        edge c_p0, c_p1, c_p2 when t_atend <= 0 do (t_next, d) := sample d goto WaitForNextBox;

    // Clocks.
\end{verbatim}
Initial position of the boxes

Listing C.15: The initial positions of the boxes.

```cpp
// These constants ensure that the starting position of each box is -1.0.
const real xinit1 = -1.0;
const real xinit2 = -1.0;
const real xinit3 = -1.0;
```
Lamp above each conveyor

Listing C.16: The hybrid automaton Lamp.

// This automaton lamp is just a simple one to use for each conveyor. Depending on the velocity of the
// conveyor the lamp turns green if the conveyor is on and turns red if the conveyor is off.

automaton def Lamp(alg int nr; alg int a_v):

    event goToOn, goToOff;

    location Off:
        initial;
        edge goToOn when a_v = 1 goto On;

    location On:
        edge goToOff when a_v = 0 goto Off;

    // SVG output of the lamp
    svgout id "lamp" + <string>nr attr "fill" value if Off: "red" else "yellowgreen" end;

end
Declaration of the mystery box

Listing C.17: The hybrid automaton MysteryBox.

import "system_perbox.cif";

// This automaton MysteryBox is a copy of the automaton Box with some small changes. The only change made is
// that the initial position xinit is not zero but defined on the top level. With the automaton it is
// possible to create UFO errors at any position of the conveyor, since a box can be added, at any point on
// the conveyor, that is not expected.

automaton def MysteryBox(alg int nr; alg real xinit; alg int a_v1, a_v2, a_v3, a_v4; uncontrollable u_pin):
    disc bool UFO = false;
    cont x = 0.0;

    location Initialize:
        equation x' = 0.0;
        initial;
        edge u_pin do x:= xinit goto Moving;

    location Moving:
        equation x' = if x < 8 : a_v1 * 1.0
                       elif x >= 8 and x < 14 : a_v2 * 1.0
                       elif x >= 14 and x < 20 : a_v3 * 1.0
                       elif x >= 20 and x < 26 : a_v4 * 1.0
                       else 1.0
        edge when x < 8 and counter1.UFO = true and not UFO = true do UFO := true;
        edge when x >= 8 and x < 14 and counter2.UFO = true and not UFO = true do UFO := true;
        edge when x >= 14 and x < 20 and counter3.UFO = true and not UFO = true do UFO := true;
        edge when x >= 20 and x < 26 and counter4.UFO = true and not UFO = true do UFO := true;
        edge when x >= 28 do UFO := false, x := -1.0
goto Initialize;

    edge u_pin;

    // Box position
    svgout id "box" + <string>nr attr "x" value 30 * x + 10.5;

    // Box visibility
svgout id "box" + <string>nr attr "visibility" value if x > -0.5: "visible" else "hidden" end;

// Change color of box if it is an UFO box
svgout id "box" + <string>nr attr "fill" value if UFO: "red" else "orange" end;

// This SVG element colors the button orange at the moment it can be pushed.
svgout id "button" + <string>nr attr "fill" value if Initialize: "orange" else "black" end;

end
Button to insert the mystery box

Listing C.18: The hybrid automaton MysteryButton.

// This automaton MysteryButton is just a simple button automaton. The button releases the Mystery box to
// fall at the conveyor and continue on it.
automaton def MysteryButton(alg int nr):

uncontrollable u_a_push, u_release;

location Push:
   edge u_a_push goto Wait;

location Wait:
   initial;
   edge u_release goto Push;
   edge u_a_push;

// SVG output:
svg id "button" + <string>nr event u_a_push;
end
Recognizing UFO and Missing events

Listing C.19: The hybrid automaton $UFO\_Miss$.

```
import "sensor_funcs.cif";

// This automaton $UFO\_Miss$ recognizes the events UFO and Missing and when an event is recognized, by a
// synchronizing event, the counter is updated. Next to this also the correct automaton is set to action, so
// the occurring event will be corrected anyway. This is done such that the current event does not appear at
// the following sensors again.

automaton def UFO_Miss(alg bool sensor; alg tuple ( real a; real b; real c; real d; real e; real f ) nom;
    alg real pos_start, pos_sens, pos_end; controllable c_corrected):

    controllable c_ufo, c_miss, c_correct_1, c_correct_2, c_correct_3;
    event potentialUFO, potentialMissing, falseAlarm, toUFOevent, toMissingEvent, toReady;

    // These three boolean variables are defined by a function out of the file sensor_funcs.cif. This functions
    // check whether the real values of the variables nom (which are the contour positions) lie in between the
    // boundaries defined as pos_start, pos_end and pos_sens. These correspond to the start and end of the
    // interval in which a contour needs to be corrected and the position of the sensor itself.
    alg bool s_nomAll = AllFramesAtSensor(nom[0], nom[1], nom[2], nom[3], nom[4], nom[5], pos_start, pos_end);
    alg bool s_nom123 = Frame123AtSensor(nom[0], nom[1], nom[2], pos_sens, pos_end);
    alg bool s_nom456 = Frame123AtSensor(nom[3], nom[4], nom[5], pos_sens, pos_end);

    // These three boolean variables are based on the same sort of function as the previous ones, only in this
    // case the three contours corresponding to one of the three boxes are considered. So not the three contours
    // which can be placed extra to a box which generates a missing error.
    alg bool s_nom_1 = xAtSensor(nom[0], pos_start, pos_end+0.1);
    alg bool s_nom_2 = xAtSensor(nom[1], pos_start, pos_end+0.1);
    alg bool s_nom_3 = xAtSensor(nom[2], pos_start, pos_end+0.1);

    // This is the initial position at which two situations can occur. The sensor can turn on, which means there
    // can be a UFO event, so the automaton is moved to the location PotentialUFO. The other situation is one of
    // the three contours of a box is entering the boundaries of correcting this contour so a potential Missing
    // error is the case.

    location Ready:
        initial;
        edge potentialUFO when sensor goto PotentialUFO;
        edge potentialMissing when s_nom123 goto PotentialMissing;
```

```
// In this location the sensor is on so a UFO event can occur, this will happen if none of the contours is at
// the "correcting interval" \( s_{nomAll} \) is false. Otherwise if one of the contours is at the interval
// \( s_{nomAll} \) is true there is a contour and it needs to be correct. Then the location NoEvent is activated.

location PotentialUFo:

edge falseAlarm when s_nomAll goto Noevent;
edge toUFOevent when not s_nomAll goto UFOevent;

// For this location the same holds as for the previous one, the only difference is that now a potential
// missing event is occurring. One of the three contours is in the interval, so if the sensor turns on their
// moment there is no longer a contour in the interval there is a missing event. At that moment the box is
// too far away from the corresponding contour.

location PotentialMissing:

edge falseAlarm when sensor goto Noevent;
edge toMissingEvent when not sensor and not s_nom123 goto Missingevent;

// These two locations makes sure when an UFO event or Missing event occurs that the other automaton like
// counter and correct error gets a signal by the synchronizing event of the event that occurs.

location UFOevent:

edge c_ufo goto Correct;

location Missingevent:

edge c_miss goto Correct;

// The following location is reached after the signal for the current event is given. In this location the
// automaton waits until it gets the synchronizing event c_corrected which means the event is corrected.

location Correct:

edge c_corrected goto BoxinSensor;

// The BoxinSensor location is reached after an event has occurred and is corrected. This because if an
// UFO event has occurred the sensor is on caused by the box, so until the sensor is off another UFO event
// cannot occur (boxes cannot be on top of each other). To avoid the automaton to recognize again a
// potential UFO event.

location BoxinSensor:

edge toMissingEvent when s_nom123 goto Missingevent;
edge toReady when not sensor goto Ready;
// When the box and a contour reach the sensor in the boundary interval the position of the contour needs to
// be corrected. This is done by the synchronizing event corresponding by the contour that is at the sensor
// at that current moment of time. By these three events the generator knows which contour to correct.

location Noevent:

    edge c_correct_1 when s_nom_1;
    edge c_correct_2 when s_nom_2;
    edge c_correct_3 when s_nom_3;

    edge toReady when not sensor goto Ready;

end
Functions used to determine the position of the boxes

Listing C.20: These functions are used to determine the box and contour positions.

// These several function are used to determine the position of boxes and/or contours at the sensors.

func bool xAtSensor(real x, pos1, pos2):
    return x >= pos1 and x <= pos2;
end

func bool Frame123AtSensor(real x1, x2, x3; real pos1, pos2 ):
    return xAtSensor(x1, pos1, pos2)
    or xAtSensor(x2, pos1, pos2)
    or xAtSensor(x3, pos1, pos2);
end

func bool BoxesAtSensor(real x1, x2, x3, x4; real pos1, pos2 ):
    return xAtSensor(x1, pos1, pos2)
    or xAtSensor(x2, pos1, pos2)
    or xAtSensor(x3, pos1, pos2)
    or xAtSensor(x4, pos1, pos2);
end

func bool AllFramesAtSensor(real x1, x2, x3, x4, x5, x6; real pos1, pos2 ):
    return xAtSensor(x1, pos1, pos2)
    or xAtSensor(x2, pos1, pos2)
    or xAtSensor(x3, pos1, pos2)
    or xAtSensor(x4, pos1, pos2)
    or xAtSensor(x5, pos1, pos2)
    or xAtSensor(x6, pos1, pos2);
end
Abstracted CIF model UFO sorter

Main file

In the Listings below all files necessary to run the abstract UFO sorter model are shown. First the main file (*system_perbox.cif*) is shown after which the remaining CIF automata are listed in alphabetical order. This abstracted model is explained in Section 5.3.1.

**Listing D.1:** Abstract CIF model that can detect UFO products and missing products. UFO’s can be detected at the exit conveyor. Missing products cause the conveyor to stop.

```plaintext
//Import of all necessary files:
svgfile "usecase_conveyors.svg";
import "generator.cif";
import "controller.cif";
import "box.cif";
import "counter.cif";
```
import "button.cif";
import "mystery_box.cif";
import "mystery_button.cif";
import "recognize_events.cif";
import "ExitConveyor.cif";

// Declaration of the four buttons to restart each conveyor after it stopped.
Abut1: AButton(1, Actrl1.av, Acounter1.c_stop_ufo);
Abut2: AButton(2, Actrl2.av, Acounter2.c_stop_ufo);
Abut3: AButton(3, Actrl3.av, Acounter3.c_stop_ufo);
Abut4: AButton(4, Actrl4.av, Acounter4.c_stop_ufo);

// Time
svgout id "time_txt" text value fmt("Time:/uni2423%.1f", time);

// Sensor colors.
svgout id "sensor1_box" attr "fill" value if As1: "green" else "red" end;
svgout id "sensor2_box" attr "fill" value if As2: "green" else "red" end;
svgout id "sensor3_box" attr "fill" value if As3: "green" else "red" end;
svgout id "sensor4_box" attr "fill" value if As4: "green" else "red" end;

// The generator.
Agen: AGeneratorStochastic(As1);

// The mysterybox and mysterybutton declaration.
Amysterybox: AMysteryBox(10, Actrl1.av, Actrl2.av, Actrl3.av, Actrl4.av, Amysterybutton.u_a_push, AeventsAt2.c_ufo, AeventsAt2.c_falseAlarm);
Amysterybutton: AMysteryButton(10, Actrl2.entering, As2, As1);

// The declaration of the sensor of the generator (checks whether a box is send).
alg bool As0 = ( Abox1.x > -1 and Abox1.x <= 3 )
  or ( Abox2.x > -1 and Abox2.x <= 3 )
  or ( Abox3.x > -1 and Abox3.x <= 3 )
  or ( Amysterybox.x > -1 and Amysterybox.x <= 3 );

// The declaration of the four sensors for each conveyor.
alg bool As1 =
( Abox1.x >= 5 and Abox1.x <= 9 )
or ( Abox2.x >= 5 and Abox2.x <= 9 )
or ( Abox3.x >= 5 and Abox3.x <= 9 )
or ( Amysterybox.x >= 5 and Amysterybox.x <= 9 );

alg bool As2 =
( Abox1.x >= 11 and Abox1.x <= 15 )
or ( Abox2.x >= 11 and Abox2.x <= 15 )
or ( Abox3.x >= 11 and Abox3.x <= 15 )
or ( Amysterybox.x >= 11 and Amysterybox.x <= 15 );

alg bool As3 =
( Abox1.x >= 17 and Abox1.x <= 21 )
or ( Abox2.x >= 17 and Abox2.x <= 21 )
or ( Abox3.x >= 17 and Abox3.x <= 21 )
or ( Amysterybox.x >= 17 and Amysterybox.x <= 21 );

alg bool As4 =
( Abox1.x >= 23 and Abox1.x <= 27 )
or ( Abox2.x >= 23 and Abox2.x <= 27 )
or ( Abox3.x >= 23 and Abox3.x <= 27 )
or ( Amysterybox.x >= 23 and Amysterybox.x <= 27 );

// Declaration of each box
Abox1: ABox(1, Actrl1.av, Actrl2.av, Actrl3.av, Actrl4.av, Agen.c_p0);
Abox2: ABox(2, Actrl1.av, Actrl2.av, Actrl3.av, Actrl4.av, Agen.c_p1);
Abox3: ABox(3, Actrl1.av, Actrl2.av, Actrl3.av, Actrl4.av, Agen.c_p2);

// Declaration of the four controllers, one for each conveyor.
Actrl1: AControl(1, As1, As0, Actrl2.stop or Actrl3.stop or Actrl4.stop, Abut1.pushed, true,
                  Agen.c_p, Acounter1.c_stop_ufo, Actrl2.c_pcout );

Actrl2: AControl(2, As2, As1, Actrl3.stop or Actrl4.stop, Abut2.pushed, Abut1.pushed,
                  Actrl1.c_pcout, Acounter2.c_stop_ufo, Actrl3.c_pcout );

Actrl3: AControl(3, As3, As2, Actrl4.stop, Abut3.pushed, Abut2.pushed,
                  Actrl2.c_pcout, Acounter3.c_stop_ufo, Actrl4.c_pcout );
Actrl4: AControl(4, As4, As3, false, Abut4.pushed, Abut3.pushed,
                  Actrl3.c_pcout, Actrl4.c_stop_ufo, Aexit.c_pcout );

Aexit: AExit(UFOincoming);

// Declaration of the automaton that recognizes events at each conveyor.
AeventsAt1: AUFOmissing(As1);

AeventsAt2: AUFOmissing(As2);

AeventsAt3: AUFOmissing(As3);

AeventsAt4: AUFOmissing(As4);

// Declaration of the counters that keep track of the events and gives a signal if an error occurs.
Acounter1: ACounter(1 , Abut1.pushed, AeventsAt1.c_ufo);
Acounter2: ACounter(2 , Abut2.pushed, AeventsAt2.c_ufo);
Acounter3: ACounter(3 , Abut3.pushed, AeventsAt3.c_ufo);
Acounter4: ACounter(4 , Abut4.pushed, AeventsAt4.c_ufo);

// Conveyor colors
svgout id "belt1_outer" attr "fill" value switch Actrl1.av:
  case 0 : "rgb(220,220,220)"
  case 1 : "rgb(33,33,33)"
  else   "rgb(33,33,33)"
end;

svgout id "belt2_outer" attr "fill" value switch Actrl2.av:
  case 0 : "rgb(220,220,220)"
  case 1 : "rgb(33,33,33)"
  else   "rgb(33,33,33)"
end;

svgout id "belt3_outer" attr "fill" value switch Actrl3.av:
case 0 : "rgb(220,220,220)"
case 1 : "rgb(33,33,33)"
else      "rgb(33,33,33)"
end;

svgout id "belt4_outer" attr "fill"
  value switch Actr14.av:
  case 0 : "rgb(220,220,220)"
  case 1 : "rgb(33,33,33)"
  else     "rgb(33,33,33)"
end;

// Displays whether the box is an UFO or not on the exit conveyor.
alg bool UFOincoming = (Amysterybox.x >= 20 and Amysterybox.UFO);

svgout id "product_type" text value if
  UFOincoming: "UFO" else "No, UFO" end;
Declaration of the boxes

Listing D.2: The abstract automaton definition $Box$. 

```plaintext
import "step.cif";

// This automaton $Box$ represents the boxes of the system, it contains the int $nr$ to distinguish the boxes from each other. $xinit$ is the initial position of the box ($-1$), $delay$ is the prescribed delay which is used to create the desired errors. $a_v$ are the different velocities of each conveyor and $c_{pin}$ is the event on which it corresponds with the generator at the moment a new box is generated.

automaton def ABox(alg int[1..4] nr; alg int[0..1] av1, av2, av3, av4; controllable c_pin):

  // This location initializes each position $x$ of a box
  location Initialize:
    initial;
    edge goToWaiting when x = -1 goto Waiting;
    edge goToFirstCnvr when x = 0 goto Moving;
    edge e_step;

  // This location initializes each position $x$ of a box
  location Waiting:
    edge c_pin do x := 0 goto Moving;
    edge e_step;

  // The position of the box is a function of the corresponding velocity of the current conveyor
  location Moving:
    edge e_step when x = 0 and av1 > 0 do x := 4;
    edge e_step when x = 0 and av1 <= 0;
    edge e_step when x = 4 and av1 > 0 do x := 5;
    edge e_step when x = 4 and av1 <= 0;
    edge e_step when x = 5 and av2 > 0 do x := 10;
    edge e_step when x = 5 and av2 <= 0;
    edge e_step when x = 10 and av2 > 0 do x := 11;
    edge e_step when x = 10 and av2 <= 0;
```
edge e_step when x = 11 and av2 > 0 do x := 16;
edge e_step when x = 11 and av2 <= 0;
edge e_step when x = 16 and av3 > 0 do x := 17;
edge e_step when x = 16 and av3 <= 0;
edge e_step when x = 17 and av3 > 0 do x := 22;
edge e_step when x = 17 and av3 <= 0;
edge e_step when x = 22 and av4 > 0 do x := 23;
edge e_step when x = 22 and av4 <= 0;
edge e_step when x = 23 and av4 > 0 do x := 28;
edge e_step when x = 23 and av4 <= 0;

edge goToStart when x >= 28 do x := -1 goto Waiting;

// Plotting the boxes at the correct position
svgout id "box" + <string>nr attr "x" value 30 * x + 10.5;

// Set color of box to green
svgout id "box" + <string>nr attr "fill" value if true: "green" else "green" end;

// Allow the box only to be visible if it is on one of the conveyors
svgout id "box" + <string>nr attr "visibility" value if x > -0.5: "visible" else "hidden" end;

end
Button in case of an error

Listing D.3: The abstract automaton definition Button.

import "step.cif";

// This is a standard automaton to control the button that needs to be pushed to restart a conveyor
// at which an error has occurred. The int nr is to distinguish the different buttons form each other, the
// events c_stop_ufo and c_stop_missing corresponds to the events from the automaton Counter in the
// counter.cif file. They occur when one of the two errors appears.

automaton def AButton(alg int nr, av; controllable c_stop_ufo):
  uncontrollable u_a_push; // the uncontrollable event to push the button
disc int[0..6] tReset = 0;

  // Event resetCounter not in hybrid model, needed to make tReset bounded.
  event autoPush, resetButton, resetClock, resetCounter;

disc bool pushed = false; // the boolean "pushed" is used to track if the button is pushed or not.

  // When a UFO occurs, the button will be pushed automatically after 1 second.
  location autoPushUFO:
    edge autoPush when tReset >= 1 do pushed := true, tReset := 0 goto Wait;
    edge e_step do tReset := tReset + STEP;

  // If there is no error, this automaton is in this location, it waits until one of the two error events
  // occur and if so, it goes again to the location Push. When it comes from the location Push into this
  // location, the boolean pushed will be set to false after 2 seconds. This is needed in order for the
  // following conveyor to go to the Running state, so that a corrected error will cause that the box can
  // start running immediately.
  location Wait:
    initial;
    edge resetButton when pushed and tReset >= 2 do pushed := false;
    edge resetClock when pushed and tReset >= 1 and av < 1 do tReset := 0;
    edge resetCounter when tReset >= 5 do tReset := 0;
    edge c_stop_ufo when not pushed do tReset := 0 goto autoPushUFO;
    edge e_step do tReset := tReset + STEP;
// Here the input of pushing the button is created.
svgIn id "button" + <string>nr event u_a_push;

// This svg element colors the button orange at the moment it can be pushed.
svgOut id "button" + <string>nr attr "fill" value if autoPushUFO: "orange" else "black" end;

end
Listing D.4: The abstract automaton definition Control.

```
import "step.cif";

// This controller is almost the same as controller 1, so here only the differences will be explained.

automaton def AControl(alg int[1..4] nr; alg bool s, sPrev, stop Else, pushed, pushedPrev; controllable c_pcin, c_stop_ufo, c_pArrivedNxtCnvr):

// The variables
// nr - just a nr given to the controller
// s - the value of the sensor at this conveyor
// stopElse - boolean variable based on a function which is true if one of the succeeding conveyors stops
// pushed - is true when the current button is pushed and is false when the pushed button is released.
// c_pcin - is the synchronizing event when a box is leaving at a preceding conveyor
// c_stop_ufo - is the synchronizing event to stop the conveyor caused by a UFO error
// c_stop_missing - is the synchronizing event to stop the conveyor caused by a missing error

c_controllable c_pcout;

disc int[0..1] av = 1;

event pLeft, stopP, cnvrDieback, goToRunning,
    errorCleared, prevButtonPushed, goToStop, pArrived;

// The variables leaving and entering represent the state of the conveyor, if a box is entering, leaving
// or both.
// The variable left represents the fact if a box has left the current conveyor or not. This value is
// true when the sensor of the current conveyor turns off and left is false if a box, probably the same
// box, is leaving at the succeeding conveyor. This variable is used to model the fact that boxes
// cannot follow each other too fast.

disc bool leaving = false;
disc bool entering = false;
disc bool left = false;

// The variables stop and dieback represents the fact if the current conveyor is in stop state or in
// dieback state. These states speaks for them self.

disc bool stop = false;
disc bool dieback = false;

disc bool didStep = false; // Indicates whether in the previous event, a time step was possible.
```
ABSTRACTED CIF MODEL UFO SORTER

location Initialize:
  initial;
  
edge goToRunning when s goto Running;
edge goToRunning when not (s or stop or stopElse) goto Running;
edge goToStop when stop do av := 0 goto Stopped;
edge cnvrDieback when stopElse do av := 0 goto DieBack;
edge e_step;

// This state is the most complicated one, it represents the conveyor when it is running. During this
// mode different situation can appear, so each line is shortly explained below.
location Running:

// An box can be send from the generator in the cases when a box is leaving or nothing is happening.
// In both cases entering will become true.
edge c_pcin when not entering do av := 1, entering := true;

// When the sensor is off and the conveyor is in leaving state,
// meaning a box has left, the variable left
// becomes true and leaving becomes false. The box has left.
edge pLeft when not s and leaving do leaving := false, left := true;

// When a box has left the current conveyor (left = true) and a box is leaving at the next conveyor
// (leavingNext) this means the box is definitely arrived, so the value of left is false again.
// c_pArrivedNxtCnvr synchronizes with edge c_pcout of the next conveyor.
edge c_pArrivedNxtCnvr when left do left := false;

// When the sensor at the current conveyor is on, a box is left but not arrived at the next conveyors
// sensor (left = true). At the same time there is no box leaving or an UFO or Missing event occurs,
// than this is the moment to stop the conveyor because otherwise boxes
// will follow each other too fast.
edge stopP when s and left and not leaving do av := 0, stop := true goto Stopped;

// When the sensor of the previous conveyor is off (not sPrev) and the current sensor is on and entering
// is still true it becomes false because a box is leaving and no box is entering.
edge pArrived when not sPrev and s and entering do entering := false;

// When an error occurs at the current conveyor these synchronizing events make sure the conveyor will
// stop and go to one of the two error states.
edge c_stop_ufo do av := 0, stop := true goto UFOError;

// When a succeeding conveyor has stopped (stopElse = true) the current conveyor needs to dieback.
edge cnvrDieback when stopElse do av := 0, dieback := true goto DieBack;

edge e_step do didStep := true;

// When the button from the preceding conveyor is pushed, // the current conveyor will start, // and to be sure this will not happen again and again, // the restriction of a time step (didStep = true) is given.
edge prevButtonPushed when pushedPrev and didStep
    do av := 1, didStep := false;

// When the sensor at the current conveyor is on, no box is leaving and no box is arriving at the next // conveyor than the box at the current conveyor can be send to the next conveyor.
edge c_pcout when s and not leaving and not left do av := 1, leaving := true, entering := false;

// This location represents the situation in which a conveyor has stopped.
location Stopped:
    // When the conveyor stopped because of time restriction it can restart again when a new box is entering.
edge c_pcin when not leaving do av := 1, leaving := false, entering := true, stop := false goto Running;

// When an error occurs at the current conveyor and this conveyor is also stopped due to the intermediate // distance the error gets preference. These synchronizing events make sure the conveyor will // stop and go to one of the two error states.
edge c_stop_ufo goto UFOError;

// When the button of the preceeding conveyor is pushed it means the error at that conveyor is repaired, // so a box is coming. (Abstract: added 'and not left' to prevent continuously switching from stopped // to running state)
edge prevButtonPushed when pushedPrev and not left do av := 1, stop := false goto Running;

edge goToRunning when s and not left do av := 1, stop := false goto Running;

// When a box is leaving at the next conveyor it has arrived at that conveyor so left can be false again.
edge c_pArrivedNxtCnvr when left do left := false;

edge e_step;

// This location represents the behavior in case of an UFO error.
location UFOError:
// When during the UFO error another box arrives at the next conveyor the left value is updated.
edge c_pArrivedNxtCnvr when left do left := false;

// When the sensor of the previous conveyor is on and no box is entering, entering becomes true.
edge c_pcin when sPrev and not entering do entering := true;

// When the button corresponding with this conveyor is pushed (pushed = true) the conveyor can start again.
// This because it is assumed that the error is cleared.
edge errorCleared when pushed and not left do stop := false, av := 1 goto Running;

edge e_step;

// The dieback situation
location DieBack:

// When non of the succeeding conveyors is stopped (not stopElse) the conveyor can start again.
edge goToRunning when not stopElse do av := 1, dieback := false goto Running;
edge e_step;

// Controller state
svgout id "ctrl" + <string>nr + "_state" text value <string>self;

end
Updating of nominal positions

Listing D.5: The abstract automaton Nominal.

```cif
import "step.cif";

// This automaton Nominal is used to correct events because the box and its contour are both in the interval
// of 0.5. This automaton also deletes contours that cause a missing event at the conveyors. By deleting these
// contours the missing event will not occur at the sensors at the following conveyors.

automaton def ANominal(alg int[1..4] nr; alg int pos1, pos2, pos3, pos4;
                          alg int av1, av2, av3, av4;
                          controllable c_pin, c_corr1, c_corr2, c_corr3,
                          c_corr4, c_del1, c_del2, c_del3, c_del14):

  event goToWaiting, goToFirstCnvr, goToStart;
  disc int[-4..28] x = -1;

  location Initialize:
    initial;
    edge goToWaiting when x = -1 goto Waiting;
    edge goToFirstCnvr when x >= 0 goto Moving;
    edge e_step;

  location Waiting:
    edge c_pin do x := 0 goto Moving;
    edge e_step;

  // The position of the contour is a function of the corresponding velocity of the current conveyor
  location Moving:
    edge e_step when x < 8 do x := x + (STEP * av1);
    edge e_step when x >= 8 and x < 14 do x := x + (STEP * av2);
    edge e_step when x >= 14 and x < 20 do x := x + (STEP * av3);
    edge e_step when x >= 20 and x < 28 do x := x + (STEP * av4);
    edge e_step when x >= 28 do x := x + (STEP * 1);
    edge goToStart when x >= 28 do x := -1 goto Waiting;

    // By these events the contour positions are corrected to the sensor positions, this because the moment
    // the frontside of the box is at a sensor can be determined exactly.
    edge c_corr1 do x := pos1 goto Corrected;
    edge c_corr2 do x := pos2 goto Corrected;
    edge c_corr3 do x := pos3 goto Corrected;
    edge c_corr4 do x := pos4 goto Corrected;
```

// With these events contours that cause a missing event can be deleted, they get the position -1.0 again.

edge c_del1 do x := -1 goto Waiting;
edge c_del2 do x := -1 goto Waiting;
edge c_del3 do x := -1 goto Waiting;
edge c_del4 do x := -1 goto Waiting;

location Corrected:

edge e_step when x < 8 do x := x + (STEP * av1);
edge e_step when x >= 8 and x < 14 do x := x + (STEP * av2);
edge e_step when x >= 14 and x < 20 do x := x + (STEP * av3);
edge e_step when x >= 20 and x < 26 do x := x + (STEP * av4);
edge goToStart when x >= 26 do x := -1 goto Waiting;

// Box position nominal
svgout id "box" + <string>nr + "_nom" attr "x" value 30 * (x) + 10.5;

// Box visibility nominal
svgout id "box" + <string>nr + "_nom" attr "visibility" value if (x) > -0.5: "visible" else "hidden" end;
end
Correcting of missing or UFO events

Listing D.6: The abstract automaton \textit{Correct}.

```
// This automaton gives the signal to correct the current event that appears in the system.
// UFO events are corrected by placing a new frame around the box, Missing events are corrected by
// deleting the current frame. By the synchronizing events c_ufo and c_miss the event that appears is known.
// Then in the current state the events c_extra or c_delete gives the signal to correct the event. c_extra
// makes sure a new frame is placed around a box with UFO error. c_delete gives the signal to delete a frame
// for which no box occurs.
automaton def ACorrect(controllable c_ufo, c_miss; alg int pos1, pos2;
          alg int[-4..28] nom1, nom2, nom3, nom4, nom5, nom6, nom7):
    controllable c_corrected, c_extra_1, c_extra_2, c_extra_3,
               c_extra_4, c_delete_1, c_delete_2, c_delete_3;
    // By these three boolean variables it is defined if an extra frame (nom4, nom5 and nom6) is at the current
    // conveyor, because there is an automaton correct for each conveyor.
    // This because if it is not at the conveyors it can be used to correct an UFO error.
    alg bool extra1 = ( nom4 >= pos1 and nom4 <= pos2 );
    alg bool extra2 = ( nom5 >= pos1 and nom5 <= pos2 );
    alg bool extra3 = ( nom6 >= pos1 and nom6 <= pos2 );
    alg bool extra4 = ( nom7 >= pos1 and nom7 <= pos2 );
    // By these three boolean variables it is defined which of the three actual frames (nom1, nom2 and nom3) is
    // at the current conveyor, because there is an automaton correct for each conveyor. If this is true the
    // system knows which frame to delete because of a missing error.
    alg bool nom1bool = ( nom1 >= pos1 and nom1 <= pos2 );
    alg bool nom2bool = ( nom2 >= pos1 and nom2 <= pos2 );
    alg bool nom3bool = ( nom3 >= pos1 and nom3 <= pos2 );
    // This is the initial location of the automaton, changing of location will appear after an missing
    // or ufo event has occured.
    location start:
    initial;
    edge c_ufo goto UFOevent;
    edge c_miss goto MissingEvent;
    // In this location an UFO error has occurred and the signal to correct this is given after it is determined
```
// which extra frame is free.
location UFOevent:
    edge c_extra_1 when extra1 = false goto Corrected;
    edge c_extra_2 when extra2 = false goto Corrected;
    edge c_extra_3 when extra3 = false goto Corrected;
    edge c_extra_4 when extra4 = false goto Corrected;

// In this location a missing error has occurred and the signal to correct this is given after it is
determined which frame is at the conveyor without a box corresponding to it.
location MissingEvent:
    edge c_delete_1 when nom1bool = true goto Corrected;
    edge c_delete_2 when nom2bool = true goto Corrected;
    edge c_delete_3 when nom3bool = true goto Corrected;

// After the correcting event has taken place the automaton comes in this Corrected location at which the
// synchronizing event c_corrected takes place, this gives the signal to the Error automaton that the error
// is corrected.
location Corrected:
    edge c_corrected goto start;
end
Counter that keeps track of missing and UFO events

Listing D.7: The abstract automaton Counter.

```plaintext
// This automaton keeps track of the history of errors. It counts which errors occurs and gives a signal
// by the synchronizing event c_stop_ufo or c_stop_missing if two type missing errors or two type UFO errors
// occur after each other. The error events are saved in a tuple called errors. There is a counter for every
// conveyor so the errors on each conveyor can be saved.

automaton def ACounter(alg int[1..4] nr; alg bool pushed; controllable c_ufo):
    controllable c_stop_ufo;
    disc int[-1..1] error0 = 0;
    disc int[-1..1] error1 = 0;
    disc int[-1..1] error2 = 0;

    disc bool UFO = false;
    event errorCleared;

    // This is the initial location in which the current event UFO or Missing is saved in the tuple and depending
    // on the error the automaton will be in the location of that current event.
    location Clean:
        initial;
        edge c_ufo do error0 := error1, error1 := error2, error2 := 1, UFO := true goto UFOError;

        // When an UFO error is recognized the automaton will be in this state, the synchronizing event c_stop_ufo
        // will stop the conveyor. Then the variable UFO becomes false and the whole process starts over again so
        // the automaton will return to the state Clean.
    location UFOError:
        edge c_stop_ufo goto UFOErrorReset;

    location UFOErrorReset:
        edge errorCleared when pushed do UFO := false goto Clean;

    // The svg element to plot the string of the first element in the tuple
    svgout id "error1" + <string>nr text value
    fmt("1./uni2423%s/uni2423", if error0 = 0: "" elif error0 = -1: "Missing" else "UFO" end);

    // The svg element to plot the string of the second element in the tuple
```
svgout id "error2" + <string>nr text value
fmt("2.\%s", if error1 = 0: "" elif error1 = -1: "Missing" else "UFO" end);

// The svg element to plot the string of the third element in the tuple
svgout id "error3" + <string>nr text value
fmt("3.\%s", if error2 = 0: "" elif error2 = -1: "Missing" else "UFO" end);
end
Creating new contours

Listing D.8: The abstract automaton Nominal_extra.

```plaintext
import "step.cif";

// This automaton Nominal_extra creates the extra frames (4, 5, 6) which are placed around a box that
// creates an UFO event. This is done such that next conveyors will not recognize this UFO event again.
// By doing this the event is corrected.
automaton def ANominalExtra(alg int[4..7] nr; alg int[0..28] pos1, pos2, pos3, pos4;
    alg int[0..1] av1, av2, av3, av4;
    controllable c_pin1, c_pin2, c_pin3, c_pin4):

event goToWaiting, goToFirstCnvr, goToStart;
disc int[-4..28] x = -1;

location Initialize:
    initial;
    edge goToWaiting when x = -1 goto Waiting;
    edge goToFirstCnvr when x >= 0 goto Moving;
    edge e_step;

// In this location the starting position of the frame is attached to the variable x. These are the positions
// of the sensors, because we can determine the moment a box is at the sensor exactly. A shortcoming of this
// is that it is not possible to drop boxes under a sensor because then the frame is placed wrong. But since
// it is assumed that boxes can not be put under the sensor this will work perfectly.
location Waiting:
    edge c_pin1 do x:= pos1 goto Moving;
    edge c_pin2 do x:= pos2 goto Moving;
    edge c_pin3 do x:= pos3 goto Moving;
    edge c_pin4 do x:= pos4 goto Moving;
    edge e_step;

location Moving:
    edge e_step when x < 8 do x := x + (STEP * av1);
    edge e_step when x >= 8 and x < 14 do x := x + (STEP * av2);
    edge e_step when x >= 14 and x < 20 do x := x + (STEP * av3);
    edge e_step when x >= 20 and x < 26 do x := x + (STEP * av4);
    edge e_step when x >= 26 and x < 28 do x := x + (STEP * 1);
```

edge goToStart when x >= 28 do x := -1 goto Waiting;

// Box position nominal
svgout id "box" + <string>nr + "_nom" attr "x" value 30 * (x) + 10.5;

// Box visibility nominal
svgout id "box" + <string>nr + "_nom" attr "visibility" value if (x) > -0.5: "visible" else "hidden" end;
end
Generating the boxes

Listing D.9: The abstract automaton \textit{GeneratorStochastic}.

```c
import "step.cif";

// This automaton describes the generator that generate boxes at the beginning of the belt. This automaton
// is for a big part based on the stochastic generator in the lecture notes, and generates boxes with in a
// stochastic behaviour.

automaton def AGeneratorStochastic(alg bool sensor1):

  controllable c_p0, c_p1, c_p2;
  controllable c_pc;

  // The SignalController sends a signal to the controller of
  // conveyor 1 that a box is coming.
  location SignalController:
    initial;
    edge c_pc when not sensor1 goto SendBox;
    edge e_step;

  // In this location the box is actually send to conveyor 1.
  location SendBox:
    edge c_p0, c_p1, c_p2 goto WaitOneTimeUnit;
    edge e_step;

  // The generator will now wait two time units before a new box can enter. This is to ensure that
  // (in the abstract model) the mysterybox is allowed to be dropped.
  location WaitOneTimeUnit:
    edge e_step goto WaitSecondTimeUnit;

  location WaitSecondTimeUnit:
    edge e_step goto SignalController;

  // Clocks.
  svgout id "t1" text value fmt("t\_next\:=\_now");

  // Generator state.
  svgout id "gen\_state" text value <string>self;
end
```
Declaration of the mystery box

Listing D.10: The abstract automaton MysteryBox.

```cif
import "step.cif";
import "system_perbox.cif";

// This automaton MysteryBox is a copy of the automaton Box with some small changes. The only change made is
// that the initial position xinit is not zero but defined on the top level. With the automaton it is
// possible to create UFO errors at any position of the conveyor, since a box can be added, at any point on
// the conveyor, that is not expected.

automaton def AMysteryBox(alg int[10..10] nr; alg int[0..1] av1, av2, av3, av4; uncontrollable u_pin;
controllable c_ufo, c_falseAlarm):

disc int[-4..28] x = -1;
disc bool UFO = false;
disc bool Missing = false;

event goToStart, BoxIsUFO;

location Initialize:
initial;
edge u_pin do x:= 10 goto Moving;
edge e_step;

location Moving:
edge e_step when x = 0 and av1 > 0 do x := 4;
edge e_step when x = 0 and av1 <= 0;
edge e_step when x = 4 and av1 > 0 do x := 5;
edge e_step when x = 4 and av1 <= 0;
edge e_step when x = 5 and av1 > 0 do x := 10;
edge e_step when x = 5 and av1 <= 0;
edge e_step when x = 10 and av2 > 0 do x := 11;
edge e_step when x = 10 and av2 <= 0;

// Added line below (not in hybrid model) so that only an UFO event can be triggered on a mysterbox
// This is necessary as in this abstracted model the UFO detection algorithm has been removed.
edge c_ufo when x = 11;
```
// A false alarm should be triggered by all boxes that are not a mysterybox
// (e.g. they will not trigger an UFO event). In other words, only if the mysterybox is at the sensor
// of the (second) conveyor, a false alarm can not be triggered.
edge c_falseAlarm when x != 11;

edge e_step when x = 11 and av2 > 0 do x := 16;
edge e_step when x = 11 and av2 <= 0;

edge e_step when x = 16 and av3 > 0 do x := 17;
edge e_step when x = 16 and av3 <= 0;

edge e_step when x = 17 and av3 > 0 do x := 22;
edge e_step when x = 17 and av3 <= 0;

edge e_step when x = 22 and av4 > 0 do x := 23;
edge e_step when x = 22 and av4 <= 0;

edge e_step when x = 23 and av4 > 0 do x := 28;
edge e_step when x = 23 and av4 <= 0;

edge BoxIsUFO when x < 8 and Acounter1.UFO and not UFO and not Missing do UFO := true;
edge BoxIsUFO when x >= 8 and x < 14 and Acounter2.UFO and not UFO and not Missing do UFO := true;
edge BoxIsUFO when x >= 14 and x < 20 and Acounter3.UFO and not UFO and not Missing do UFO := true;
edge BoxIsUFO when x >= 20 and x < 26 and Acounter4.UFO and not UFO and not Missing do UFO := true;

edge goToStart when x >= 28 do UFO := false, Missing := false, x := -1
goto Initialize;

// Box position
svgout id "box" + <string>nr attr "x" value 30 + x + 10.5;

// Box visibility
svgout id "box" + <string>nr attr "visibility" value if x > -0.5: "visible" else "hidden" end;

// Change color of box if it is an UFO box
svgout id "box" + <string>nr attr "fill" value if UFO: "red" else "orange" end;

// This SVG element colors the button orange at the moment it can be pushed.
svgout id "button" + <string>nr attr "fill" value if Initialize: "orange" else "black" end;
end
Button to insert the mystery box

Listing D.11: The abstract automaton *MysteryButton*.

```java
// This automaton MysteryButton is just a simple button automaton. The button releases the Mystery box to
// fall at the conveyor and continue on it.
automaton def AMysteryButton(alg int[10..10] nr; alg bool convEntering, s, sPrev):

  uncontrollable u_a_push, u_release;

  location Push:
    edge u_a_push when not convEntering and not s and not sPrev goto Wait;

  location Wait:
    initial;
    edge u_release goto Push;

  // SVG output:
  svgin id "button" + <string>nr event u_a_push;

end
```
Recognizing UFO and Missing events

Listing D.12: The abstract automaton UFO_Missing.

```plaintext
// This automaton UFO_Missing recognizes the events UFO and Missing and when an event is recognized, by a
// synchronizing event, the counter is updated. Next to this also the correct automaton is set to action, so
// the occurring event will be corrected anyway. This is done such that the current event does not appear at
// the following sensors again.

automaton def AUFOmissing(alg bool sensor):

    controllable c_ufo, c_corrected, c_falseAlarm;
    event potentialUFO, toReady;

    // This is the initial position at which two situations can occur. The sensor can turn on, which means there
    // can be a UFO event, so the automaton is moved to the location PotentialUFO. The other situation is one of
    // the three contours of a box is entering the boundaries of correcting this contour so a potential Missing
    // error is the case.
    location Ready:
        initial;
        edge potentialUFO when sensor goto UFOevent;

        // These two locations makes sure when an UFO event or Missing event occurs that the other automaton like
        // counter and correct error gets a signal by the synchronizing event of the event that occurs.
        location UFOevent:
            edge c_ufo goto Correct;
            edge c_falseAlarm goto Noevent;

            // The following location is reached after the signal for the current event is given. In this location the
            // automaton waits until it gets the synchronizing event c_corrected which means the event is corrected.
            location Correct:
                edge c_corrected goto BoxinSensor;

                // The BoxinSensor location is reached after an event has occurred and is corrected. This because if an
                // UFO event has occurred the sensor is on caused by the box, so until the sensor is off another UFO event
                // cannot occur (boxes cannot be on top of each other). To avoid the automaton to recognize again a
                // potential UFO event.
                location BoxinSensor:
```
Step size declaration

**Listing D.13:** Declaration of the step size and step event.

```plaintext
const int STEP = 1;
event e_step;
```
Tick automaton that simulates urgency

Listing D.14: The abstract automaton tick that simulates urgency. In Section 3.5 it is explained how this large automaton is obtained.

```
import "system_perbox.transformed.cif";

automaton Tick:
    location One:
        initial;

    edge e_step when not 

    ( ( Abox1.Initialize and Abox1.x >= 0 or (Abox1.Waiting and false or Abox1.Moving and false)
    ) or ( Abox1.Initialize and false or (Abox1.Waiting and false or Abox1.Moving and false)
    ) or ( Abox1.Initialize and Abox1.x = -1 or (Abox1.Waiting and false or Abox1.Moving and false)
    ) or ( Abox2.Initialize and Abox2.x >= 0 or (Abox2.Waiting and false or Abox2.Moving and false)
    ) or ( Abox2.Initialize and false or (Abox2.Waiting and false or Abox2.Moving and false)
    ) or ( Abox2.Initialize and Abox2.x = -1 or (Abox2.Waiting and false or Abox2.Moving and false)
    ) or ( Abox3.Initialize and Abox3.x >= 0 or (Abox3.Waiting and false or Abox3.Moving and false)
    ) or ( Abox3.Initialize and false or (Abox3.Waiting and false or Abox3.Moving and false)
    ) or ( Abox3.Initialize and Abox3.x = -1 or (Abox3.Waiting and false or Abox3.Moving and false)
    ) or ( Abut1.autoPushUFO and Abut1.tReset >= 1 or Abut1.Wait and false
    ) or ( Abut1.autoPushUFO and false or Abut1.Wait and (Abut1.pushed and Abut1.tReset >= 2)
    ) or ( Abut1.autoPushUFO and false or Abut1.Wait and Abut1.pushed and (Abut1.tReset >= 1
    and Abut1.av < 1)
    ) or ( Abut1.autoPushUFO and false or Abut1.Wait and Abut1.tReset >= 5
    ) or ( Abut2.autoPushUFO and Abut2.tReset >= 1 or Abut2.Wait and false
    ) or ( Abut2.autoPushUFO and false or Abut2.Wait and (Abut2.pushed and Abut2.tReset >= 2)
    ) or ( Abut2.autoPushUFO and false or Abut2.Wait and Abut2.pushed and (Abut2.tReset >= 1
    and Abut2.av < 1)
    ) or ( Abut2.autoPushUFO and false or Abut2.Wait and Abut2.tReset >= 5
    ) or ( Abut3.autoPushUFO and Abut3.tReset >= 1 or Abut3.Wait and false
    ) or ( Abut3.autoPushUFO and false or Abut3.Wait and (Abut3.pushed and Abut3.tReset >= 2)
    ) or ( Abut3.autoPushUFO and false or Abut3.Wait and Abut3.pushed and (Abut3.tReset >= 1
    and Abut3.av < 1)
    ) or ( Abut3.autoPushUFO and false or Abut3.Wait and Abut3.tReset >= 5
```
or ( Abut4.autoPushUFO and Abut4.tReset >= 1 or Abut4.Wait and false)
or ( Abut4.autoPushUFO and false or Abut4.Wait and (Abut4.pushed and Abut4.tReset >= 2))
or ( Abut4.autoPushUFO and false or Abut4.Wait and Abut4.pushed and (Abut4.tReset >= 1 and Abut4.av < 1))
or ( Abut4.autoPushUFO and false or Abut4.Wait and Abut4.tReset >= 5)
or ( (Abut1.autoPushUFO and false or Abut1.Wait and not Abut1.pushed) and ((Acounter1.Clean and false or (Acounter1.UFOError and true or Acounter1.UFOErrorReset and false)) and (Actrl1.Initialize and false or Actrl1.Running and true or (Actrl1.Stopped and true or (Actrl1.UFOError and false or Actrl1.DieBack and false))))
or ( Acounter1.Clean and false or (Acounter1.UFOError and false or Acounter1.UFOErrorReset and Acounter1.pushed))
or ( (Abut2.autoPushUFO and false or Abut2.Wait and not Abut2.pushed) and ((Acounter2.Clean and false or (Acounter2.UFOError and true or Acounter2.UFOErrorReset and false)) and (Actrl2.Initialize and false or Actrl2.Running and true or (Actrl2.Stopped and true or (Actrl2.UFOError and false or Actrl2.DieBack and false))))
or ( Acounter2.Clean and false or (Acounter2.UFOError and false or Acounter2.UFOErrorReset and Acounter2.pushed))
or ( (Abut3.autoPushUFO and false or Abut3.Wait and not Abut3.pushed) and ((Acounter3.Clean and false or (Acounter3.UFOError and true or Acounter3.UFOErrorReset and false)) and (Actrl3.Initialize and false or Actrl3.Running and true or (Actrl3.Stopped and true or (Actrl3.UFOError and false or Actrl3.DieBack and false))))
or ( Acounter3.Clean and false or (Acounter3.UFOError and false or Acounter3.UFOErrorReset and Acounter3.pushed))
or ( (Abut4.autoPushUFO and false or Abut4.Wait and not Abut4.pushed) and ((Acounter4.Clean and false or (Acounter4.UFOError and true or Acounter4.UFOErrorReset and false)) and (Actrl4.Initialize and false or Actrl4.Running and true or (Actrl4.Stopped and true or (Actrl4.UFOError and false or Actrl4.DieBack and false))))
or ( Acounter4.Clean and false or (Acounter4.UFOError and false or Acounter4.UFOErrorReset and Acounter4.pushed))
or ( (Actrl1.Initialize and false or Actrl1.Running and Actrl1.s and (not Actrl1.leaving and not Actrl1.left)) or (Actrl1.Stopped and false or (Actrl1.UFOError and false or Actrl1.DieBack and false)))
or (Actrl2.Initialize and false or Actrl2.Running and not Actrl2.entering or (Actrl2.Stopped and not Actrl2.leaving or (Actrl2.UFOError and (Actrl2.sPrev and not Actrl2.entering) or Actrl2.DieBack and false)))
or ( Actrl1.Initialize and false or Actrl1.Running and false or (Actrl1.Stopped and false or (Actrl1.UFOError and false or Actrl1.DieBack and false)))
or ( Actrl1.Initialize and (Actrl1.s or not(Actrl1.s or Actrl1.stop or Actrl1.stopElse)) or Actrl1.Running and false or (Actrl1.Stopped and (Actrl1.s and not Actrl1.left) or (Actrl1.UFOError and false or Actrl1.DieBack and not Actrl1.Running and false) or (Actrl1.Stopped and false or (Actrl1.UFOError and false or Actrl1.DieBack and false)))
(Actrl4.Initialize and false or Actrl4.Running and Actrl4.s and (Actrl4.left and not Actrl4.lying) or (Actrl4.Stopped and false or (Actrl4.UFOError and false or Actrl4.DieBack and false))) or (AeventsAt1.Ready and false or AeventsAt1.UFOevent and false or (AeventsAt1.Correct and true or (AeventsAt1.BoxinSensor and false or AeventsAt1.Noevent and false))) or (AeventsAt1.Ready and false or AeventsAt1.UFOevent and true or (AeventsAt1.Correct and false or (AeventsAt1.BoxinSensor and false or AeventsAt1.Noevent and false))) or (AeventsAt1.Ready and false or AeventsAt1.UFOevent and false or (AeventsAt1.Correct and false or (AeventsAt1.BoxinSensor and true or AeventsAt1.Noevent and false))) or (AeventsAt1.Ready and AeventsAt1.sensor or AeventsAt1.UFOevent and false or (AeventsAt1.Correct and false or (AeventsAt1.BoxinSensor and false or AeventsAt1.Noevent and false))) or (AeventsAt1.Ready and false or AeventsAt1.UFOevent and false or (AeventsAt1.Correct and false or (AeventsAt1.BoxinSensor and not AeventsAt1.sensor or AeventsAt1.Noevent and not AeventsAt1.sensor))) or (AeventsAt2.Ready and false or AeventsAt2.UFOevent and false or (AeventsAt2.Correct and true or (AeventsAt2.BoxinSensor and false or AeventsAt2.Noevent and false))) or (AeventsAt2.Ready and false or AeventsAt2.UFOevent and true or (AeventsAt2.Correct and false or (AeventsAt2.BoxinSensor and false or AeventsAt2.Noevent and false))) and (Amysterybox.Initialize and false or Amysterybox.Moving and Amysterybox.x = 11)) or (Acounter2.Clean and true or (Acounter2.UFOError and false or Acounter2.UFOErrorReset and false)) and (AeventsAt2.Ready and false or AeventsAt2.UFOevent and true or (AeventsAt2.Correct and false or (AeventsAt2.BoxinSensor and false or AeventsAt2.Noevent and false))) or (AeventsAt2.Ready and AeventsAt2.sensor or AeventsAt2.UFOevent and false or (AeventsAt2.Correct and false or (AeventsAt2.BoxinSensor and false or AeventsAt2.Noevent and false))) or (AeventsAt2.Ready and false or AeventsAt2.UFOevent and false or (AeventsAt2.Correct and false or (AeventsAt2.BoxinSensor and not AeventsAt2.sensor or AeventsAt2.Noevent and not AeventsAt2.sensor))) or (AeventsAt3.Ready and false or AeventsAt3.UFOevent and false or (AeventsAt3.Correct and true or (AeventsAt3.BoxinSensor and false or AeventsAt3.Noevent and false))) or (AeventsAt3.Ready and false or AeventsAt3.UFOevent and true or (AeventsAt3.Correct and false or (AeventsAt3.BoxinSensor and false or AeventsAt3.Noevent and false))) or (AeventsAt3.Ready and false or AeventsAt3.UFOevent and false or (AeventsAt3.Correct and false or (AeventsAt3.BoxinSensor and false or AeventsAt3.Noevent and false))) or (AeventsAt3.Ready and AeventsAt3.sensor or AeventsAt3.UFOevent and false or (AeventsAt3.Correct and false or (AeventsAt3.BoxinSensor and false or AeventsAt3.Noevent and false))) or (AeventsAt3.Ready and false or AeventsAt3.UFOevent and false or (AeventsAt3.Correct and false or (AeventsAt3.BoxinSensor and false or AeventsAt3.Noevent and false))) or (AeventsAt4.Ready and false or AeventsAt4.UFOevent and false or (AeventsAt4.Correct and true or (AeventsAt4.BoxinSensor and false or AeventsAt4.Noevent and false)))
end
CIF specification of single conveyor UFO sorter model

The code for the model can be found on the next page. A visualization of the automata of the model can be found in Section E (‘Automata visualization’, last section of this appendix). A short explanation of the model can be found in Section 5.4.1.
Main file

Listing E.1: The main file of the specification of the single conveyor UFO sorter model.

```plaintext
// Import of all necessary files:
svgfile "../usecase_conveyors.svg";
import "generatorNoEstep.cif";
import "../mystery_button.cif";
import "../ExitConveyor.cif";
import "MultipleConveyors.cif";

// Entry
Agen : AGeneratorStochastic(Cnvr1.s);
AS0 : Sensor0(Agen.c_pc, Aexit.c_pcout, Cnvr1.c_pout_UFO);

// Specification
Cnvr1 : UniConveyor(Agen.c_pc, Amysterybutton.u_a_push,
                     Aexit.c_pcout, Aexit.c_UFOincomingEvent);
TrckMysterybox : TrackMysterybox(Amysterybutton.u_a_push, Cnvr1.c_pout_UFO,
                                   Aexit.c_reset_UFOdetected, Cnvr1.e_turnOff_UFO);
goToStartNonUFO : goToStartNonUFOHandling(Aexit.c_pcout,
                                          Cnvr1.e_toAtSensor, Cnvr1.e_turnOff_UFO);

// Displays whether the box is an UFO or not on the exit conveyor.
alg bool UFOincoming = Cnvr1.UFO;

// Button for generating UFO
Amysterybutton : AMysteryButton(10, Cnvr1.entering, Cnvr1.s, AS0.sensor0);

// Exit
Aexit : AExit(UFOincoming);
```
Listing E.2: The ‘MultipleConveyors.cif’ file. This file models the universal conveyor.

// This automaton simulates the behaviour of sensor AS0 in the implementation. // The sensor in the implementation is a function of the variables ‘x’ of the boxes, // which is not present anymore in the specification. // The sensor turns on when a product is send by the generator, and turns off when a product arrived // at the next sensor (sensor1).

automaton def Sensor0(controllable c_pc; event e_pout, e_pout_UFO):
  disc bool sensor0 = false;

location SensorOff:
  initial;
  edge c_pc do sensor0 := true goto SensorOn;

location SensorOn:
  edge e_pout_UFO do sensor0 := false goto SensorOff;
  edge e_pout do sensor0 := false goto SensorOff;
end

automaton def UniConveyor(event e_pin, e_UFO_in, e_pout; controllable c_UFOincomingEvent):
  event e_toAtSensor, e_turnOff_UFO;
  controllable c_pout_UFO;
  disc bool UFO = false;
  // In the implementation an UFO can only be detected at the sensor, therefore // only in state ‘Entering’ the UFO variable can be set to entering. // In state ‘empty’ we therefore use the variable ‘possibleUFO’.
  disc bool possibleUFO = false;
  disc bool s = false;
  disc bool entering = false;

location Empty:
  initial;
  edge e_pin do entering := true goto Entering;
  // When an UFO is dropped on the conveyor, the controller of this conveyor will not be aware of this // as the UFO has not hit the sensor and thus the state of the controller remains unchanged.
  edge e_UFO_in do possibleUFO := true;

location Entering:
edge e_toAtSensor do entering := false, s := true goto Occupied;

location Occupied:
edge e_pout when not possibleUFO do s := false goto Empty;
edge c_pout_UFO when possibleUFO do s := false, UFO := true goto WaitUFOincomingEvent;

location WaitUFOincomingEvent:
edge c_UFOincomingEvent goto UFOexit;

location UFOexit:
edge e_turnOff_UFO do UFO := false, possibleUFO := false goto Empty;
end

automaton def TrackMysterybox(uncontrollable u_a_push; controllable c_pout_UFO, c_reset_UFOdetected; event e_turnOff_UFO):

location WaitPushUFO:
initial;
edge u_a_push goto UFOLeaving;

location UFOLeaving:
edge c_pout_UFO goto UFOTrue;

location UFOTrue:
edge e_turnOff_UFO goto resetUFO;

location resetUFO:
edge c_reset_UFOdetected goto WaitPushUFO;
end

// This automata ensure that a goToStartNonUFO event happens after an e_pout/c_UFOincomingEvent, // and before e_toAtSensor.
// A c_UFOincomingEvent event means that a UFO product is nearing the exit conveyor, // and a non-UFO product is behind the UFO product.
automaton def goToStartNonUFOHandling(event e_pout, e_toAtSensor, e_turnOff_UFO):
event goToStartNonUFO;

location ONE:
initial;
edge e_toAtSensor goto TWO;

location TWO:
edge e_pout goto THREE;
edge e_turnOff_UFO goto THREE;

location THREE:
edge goToStartNonUFO goto ONE;
end
Listing E.3: The ‘generatorNoEstep.cif’ file. This file models the generator.

// This automaton describes the generator that generate boxes at the beginning of the belt. This automaton
// is for a big part based on the stochastic generator in the lecture notes, and generates boxes with in a
// stochastic behaviour.

automaton def AGeneratorStochastic(alg bool sensor1):

    controllable c_p0;
    controllable c_pc;

    // The SignalController sends a signal to the controller of
    // conveyor 1 that a box is coming.
    location SignalController:
        initial;
        edge c_pc when not sensor1 goto SendBox;

    // In this location the box is actually send to conveyor 1.
    location SendBox:
        edge c_p0 goto SignalController;

    // Clocks.
    svgout id "t1" text value fmt("t_next::\now" );

    // Generator state.
    svgout id "gen_state" text value <string>self;

end
Listing E.4: The ‘mystery_button.cif’ file. This file models the UFO drop button.

// This automaton MysteryButton is just a simple button automaton. The button releases the Mystery box to
// fall at the conveyor and continue on it.
automaton def AMysteryButton(alg int[10..10] nr; alg bool convEntering, s, sPrev):

uncontrollable u_a_push, u_release;

location Push:
    edge u_a_push when not convEntering and not s and not sPrev goto Wait;

location Wait:
    initial;
    edge u_release goto Push;

// SVG output:
svin id "button" + <string>nr event u_a_push;

end
Exit conveyor

Listing E.5: The ‘ExitConveyor.cif’ file. This file models the exit conveyor.

```cif
// This is the exit conveyor that will always accept incoming products
automaton def AExit(alg bool UFOincoming):
  controllable c_pcout, c_UFOincomingEvent, c_reset_UFOdetected;
  disc bool UFOdetected = false;

location One:
  initial;
  edge c_pcout;

  // c_UFOincomingEvent ensures an event is visible after UFOincoming becomes true
  edge c_UFOincomingEvent when UFOincoming and not UFOdetected do UFOdetected := true;

  // The event below will reset UFOdetected once the UFO disappeared again.
  edge c_reset_UFOdetected when not UFOincoming and UFOdetected do UFOdetected := false;
end
```
Automata visualization

A visualization of the automata can be found on the next page.
Bibliography


