Design of a Model-Based Controller for Baggage Handling Systems with Data Tracking

Master's Thesis

T.S. Zwijgers
0717911
CST 2015.033
March 2015
Design of a Model-Based Controller for Baggage Handling Systems with Data Tracking

T.S. Zwijgers (0717911)

CST 2015.033 NON-CONFIDENTIAL

Master’s thesis

Supervisor:  Prof.dr. W.J. Fokkink
Advisors:    Dr.ir. D.A. van Beek
            Ir. L. Swartjes
            Dr.ir. J.A.W.M. van Eekelen (Vanderlande)
            Ir. R. Hommels (Vanderlande)

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF MECHANICAL ENGINEERING
SYSTEMS ENGINEERING GROUP

March 2015
Graduation project
Integration of a communication protocol in a controller for a baggage handling system

Subject
Vanderlande is a manufacturer of material handling systems, such as baggage handling systems at airports. These systems are controlled by Programmable Logic Controllers (PLCs). The software of these PLCs consists of different levels and modules. For each control level, controllers have to be developed that contain the desired functionalities, specified by the (informal) requirements in the product book. For this project an experimental setup of a baggage handling system is available. This setup consists of multiple sorting, merging and transportation zones. In previous projects, controllers have been developed for this setup using CIF. This tool allows for early validation of controllers by visualization using a CIF specified model of the hardware. Controllers created in CIF can be transformed to PLC code and implemented on the experimental setup. As a next step towards industrial application with multiple control levels, a standard for data communication and data transferring between levels is required. This standard should make it possible to use the Vanderlande “LIC record”, the list of required actions for each product, to guide products through the system to their destination.

Assignment
• Develop and implement supervisory control models according to the requirements for the Check-in Desk, Collector, Carrousel (triplanar) and Identification Zone.

• Investigate and implement the possibility for data transfer between controller levels with the goal of guiding products through the system. Investigate which information is required and at which times communication takes place in the current system.

• Improve the existing models and controllers with the new functionalities introduced in CIF.

• Take into account the CIF to PLC code transformation and consider the modularity, readability and efficiency of the transformed code.

Where innovation starts
Start April 2014
Finish January 2015
Student T.S. Zwijgers
Supervisors prof.dr. W.J. Fokkink, Dr.ir. D.A. van Beek, Ir. L. Swartjes, Dr.ir. J.A.W.M. van Eekelen (Vanderlande), Ir. R. Hommels (Vanderlande)
Preface

This report marks the end of a year-long graduation project, and the final step to my Master’s degree. In 2012 I gained my Bachelor’s degree in Mechanical Engineering at the Eindhoven University of Technology after three years of study. In that same year I started with a Master’s degree in Mechanical Engineering with the research groups Manufacturing Networks and Systems Engineering. I had enjoyed my Bachelor studies, but felt like I had not yet experienced what the technological industry was like outside of the university. I did my internship in Gardanne, France at the Centre Micro-Électronique De Provence in collaboration with ST Micro Electronics under the supervision of Pascal Etman, which was a great experience. However, for my final project I wanted to work more closely with a company than I did for my internship. Furthermore, my internship was typically a Manufacturing Networks assignment, and I wanted to try a project with Systems Engineering. I contacted Bert van Beek, who proposed an assignment with Vanderlande. After a short initial meeting I decided to do a project in the framework of the SUCCESS project of Lennart Swartjes, starting in March 2014. Now, almost a year later I am almost finished and have seen the entire cycle from a problem to an implemented solution.

At the university I would like to thank Bert van Beek for the advice and insightful comments during all, sometimes impromptu, meetings. I would like to thank Lennart Swartjes for always having time for questions and the quick responses during the final testing. I also would like to thank Dennis Hendriks for his great support with the CIF tooling.

At Vanderlande I would like to thank everybody involved with the SUCCESS project. Particularly those who attended my Progress Meetings for their advice and participation in the discussions. Furthermore, I would like to thank Joost van Eekelen for his support and René Hommels for his great enthusiasm.

Finally I would like to thank my parents and sisters for their support during this project and my entire studies.

Tom Zwijgers
Eindhoven, March 2014
Abstract

Vanderlande is a manufacturer of Baggage Handling Systems (BHSs) at airports. Typically, BHSs are controlled using Programmable Logic Controllers (PLCs). Currently, each brand of PLC has its own distinct programming language and often there are region-specific preferences regarding PLC brands. Vanderlande operates internationally and has to deal with manually recoding programs between brands. As a solution, they are looking for a software environment capable of generating code from a source model to other platforms. Furthermore, in order to shorten the design process and reduce the numbers of programming errors, this new environment should support formal requirements and early verification of correctness. CIF is a language in development at the Systems Engineering Group at the Eindhoven University of Technology. It is based on hybrid automata and supports code generation from a CIF model to other languages. The automata based structure makes it possible to use formal requirements. Moreover, the on-board interactive visualization can be used for early testing of control software.

Previous projects created controller models in CIF, these were transformed to a PLC program and used to control a hardware loop. A library of control definitions was created in keeping with the Vanderlande Area-Zone-Section architecture. By instantiating these definitions, BHS control systems can be assembled. This project continues with extending the control library in order to control an entire airport BHS. This requires the implementation of a set of information for each product, a baggage record (BR). Furthermore, Zones and Sections handling input, output and measurement are introduced. Because of newly introduced CIF functionality, the existing control library is extended and reevaluated.

In order to hold and update a BR for each product, a globally accessible database is introduced in the form of an automaton holding an array of tuples. This database is referred to as the BR Array. Each tuple represents the BR for one product. BRs are updated with information sent by measuring processes via channels. Using an interfacing process, communication with measuring equipment manufactured by outside suppliers is introduced. This process handles all signals and messages to and from the measuring object. In the case of new information requiring an update to a BR, the interfacing process uses the update channel for that update type to relay the information to the BR Array.

The input, output and measurement zones were designed according to the control requirements listed in the Vanderlande product books. The measuring zones use the interfacing
processes to interact with the measuring equipment. Existing control definitions have
been reevaluated in terms of scalability, interfacing and reuse. Zone and section con-
trollers should be scalable and applicable in most situations. However, when introducing
extra or optional functionality, the impact on the definition should be small. If this is
not possible, creating a new and separate definition for that function variant should be
considered.

The BR database and the control library have been tested with the interactive visualiza-
tion. A suitable test setup is chosen in the form of an existing emulation model. The
specifications of this setup are used to create a BHS controller with the definition library.
This BHS controller is transformed and implemented on the PLC with Siemens TIA Portal.
Using an emulated BHS, the control software can be operated on the PLC and tested.
The first step is to execute a debugging phase of the program. The errors found during
this debugging phase could be categorized in three groups.

- Errors made in parameterizing the instantiations in CIF and mapping the I/O
  variables in the PLC
- Incorrect behaviour introduced by the transformation
- Modeling errors in CIF

Errors in the model definitions occur very rarely, showing the effectiveness of the inter-
active visualization. The second step is to formally test the control software. For the
testing phase, a set of requirements is compiled to form a test plan. Two options for rep-
resenting requirements with automata were considered. It was found that a requirement
describing how it should work usually strongly resembled the actual implementation
of that requirement. A second method of formulation used automata to prove when
a requirement failed. The test plan is carried out in the CIF environment first. The
validated CIF software is transformed to the PLC where the test plan is reiterated. All
errors encountered on the PLC could be traced back to the transformation. This indicates
that a property-preserving and complete transformation would transform validated CIF
controller models to a validated PLC program.

This project has shown that using CIF, an entire BHS can be controlled. Further research
could focus on:

- Automatic generation and instantiation of CIF programs.
- Automatic testing using the CIF simulator.
- Direct industrial application of a CIF generated controller.

An example of direct industrial application could be project-specific sorting zones which
are not standardized and run on a separate PLC.
# Contents

<table>
<thead>
<tr>
<th>Preface</th>
<th>iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>vii</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>viii</td>
</tr>
</tbody>
</table>

## 1 Introduction

1. Background
   1.1 Vanderlande
   1.2 Baggage Handling Systems
   1.3 Compositional Interchange Format
   1.4 Programmable Logic Controllers
   1.5 Code Generation

1.2 Preceding Work  
1.3 Related Work  
1.4 Goal  
1.5 Outline

## 2 Modeling in CIF

2.1 Automata and Events
   2.1.1 Named Events
   2.1.2 Channels

2.2 Variable Classes

2.3 Definitions

2.4 Interactive Visualization

## 3 Data Communication

3.1 Product Information
   3.1.1 Form of the Baggage Record
   3.1.2 Location of Product Information
   3.1.3 Form of the Database
   3.1.4 Product Identity

3.2 Object Communication
   3.2.1 Identification Zone
# Contents

4 **Scaling, Interfacing and Reuse** ........................................... 31
   4.1 Scalability ........................................................................ 31
      4.1.1 Scalable Tracking Size ........................................... 31
      4.1.2 Multiple PEC Sensors ............................................ 34
   4.2 Interfacing ......................................................................... 37
      4.2.1 Multiple In- and Outfeeds ....................................... 37
      4.2.2 Window Inshooting .................................................. 43
   4.3 Reuse ................................................................................. 45
      4.3.1 Bi-Directional Conveyors ....................................... 45
      4.3.2 Master and Slave Configurations ............................. 48

5 **Real Time Implementation** ............................................... 51
   5.1 Controlling a Small Airport ............................................. 51
      5.1.1 Modeling the In- and Output Zones .......................... 51
      5.1.2 Airport ...................................................................... 55
      5.1.3 CIF Control Software ............................................. 56
   5.2 Transformation and Implementation .................................. 58
      5.2.1 Transformation ........................................................ 58
      5.2.2 Siemens TIA Portal .................................................. 59
   5.3 Emulation Testing ............................................................ 61
      5.3.1 Emulation and PLC ................................................. 61
      5.3.2 Debugging the Models and the Transformation ......... 62
      5.3.3 Connecting to Objects ............................................ 65
   5.4 Testing and Performance ................................................... 67
      5.4.1 Test Plan ................................................................. 67
      5.4.2 Test Report ............................................................. 70
      5.4.3 Analysis of System Performance and the PLC Code ..... 71

6 **Conclusions and Recommendations** ................................. 73
   6.1 Conclusions ...................................................................... 73
   6.2 Recommendations ........................................................... 75

List of Abbreviations .................................................................. 79

Bibliography .............................................................................. 81

A Test Phase (Confidential) ............................................................ 85

B Vanderlande Terms and Sources (Confidential) ........................ 87

C The BR Tuple (Confidential) ...................................................... 89
Chapter 1

Introduction

1.1 Background

In recent years the demand for large industrial systems is growing, while the required solutions are becoming more complex. This has raised new challenges for the industry. In designing the control software, higher levels of cooperation between different disciplines are required.

Issues arise with for instance, defining system requirements. Textual requirements can be ambiguously worded or be open for interpretation. Furthermore, validating whether these requirements are met is expensive and time consuming. Running tests requires a hardware setup or at least an emulation model and can be performed only relatively late in the design process.

Furthermore, there is no general consensus on the control platform used, which often differs between projects. The control software for a programmable logic controller (PLC) is different from that of an industrial PC. In the case of PLCs there are also different brands, leading to even more variations.

1.1.1 Vanderlande

Vanderlande is an internationally operating manufacturer of automated material handling systems, specializing in baggage handling systems, parcel and postal sorting systems and automated warehousing. They are based in Veghel, the Netherlands, but operate all around the world.

As each customer comes with a custom problem, each project is unique and customized to the wishes and requirements of the customer. This does not mean that Vanderlande has to start over with every project. While the layout and specifics are unique to each project, the control of the individual parts e.g. a conveyor, is often largely unchanged.
In baggage handling systems where PLCs are uniformly used for control, this means that they build these systems from existing functions blocks in a library. However, in the airport industry each region often has a preference for a certain brand of PLC and different brands use different programming languages. As a result, Vanderlande needs to employ specialists on different kinds of PLCs and software needs to be available on all the different platforms.

The design cycle employed by Vanderlande makes a distinction between the domain engineer who defines the control requirements and the software engineer who implements these requirements. By using this construction, it is ensured that software has the same behaviour even when using different programming platforms and as such different software engineers. However, by using an informal syntax for these requirements, differences and errors can be introduced due to different interpretations.

In order to solve these issues, Vanderlande is experimenting with new methods of developing software. An environment in which one central control model could be ported to different platforms using an automatic code generation would eliminate manual recoding. Furthermore, Vanderlande would be more prepared for the future, as software for new emerging control platforms can be generated from the same central model.

Such an environment should also have a formal way of specifying requirements. Using for instance automata, requirements can be formally expressed. This way, requirements are easily and quickly understood and there is no room for interpretation. Furthermore, validating the correctness of these requirements in the software could greatly simplify the testing process and help catch errors early on.

1.1.2 Baggage Handling Systems

In every airport with passenger air travel, a baggage handling systems or BHS can be found. It is a network of conveyor belts, sorting and merging equipment which guides the passenger’s luggage from the check-in desk to the right destination. For an airport to function, it is of the utmost importance that the BHS is operational. For this reason, the system is often designed to have multiple redundant routing possibilities. In order to meet these requirements, the control software should be stable and error-free.

The BHS is controlled using a two-layer system. The high-level controls (HLC) layer consists of a number of modules with different functions. These modules keep track of hardware status, set destinations, control routing tables and so on. The low-level controls (LLC) layer directly controls the hardware and is distributed over a number of PLCs. Depending on the size of the BHS, this can range from one to a hundred PLCs.

Vanderlande uses the terminology and system architecture shown in Figure 1.1.
1.1. Background

A section is a single piece of hardware performing a certain function. The most commonly used section is a Transport Section, abbreviated as TRS, which is a standard conveyor. Furthermore, switch sections (SWSs) and vertibelts (VBs) are sections used in sorting equipment. The VB is a conveyor arm, it can be extended or retracted using the SWS as a switching motor.

A combination of sections is a zone. In a zone, these sections cooperate to perform some function. The simplest example of a zone is a transport zone (TRZ) which consists of a several consecutive TRSs. A SWS, a VB and a TRS on which they deposit diverted products, form a vertibelt zone (VBZ). A VBZ is shown in Figure 1.2. There are many different zones each having their own specific function.
Above zones there are areas. An area is a number of zones running on a single PLC. Areas can have a specific function or contain more than one type of functionality, depending on the total size of the system.

Elements are subparts of a section, usually having to do with inputs or outputs. For instance, in the case of the TRS, these are a Photo Electric Cell (PEC), a motor and the Local Motor Starter (LMS). The PEC is a sensor detecting the presence of products. The LMS is the interface for the sensor and actuator signals between the TRS and the PLC.

Objects are a special case which are not always integrated in the same way. An object is a piece of equipment which operates independently from the total system. It receives a request, performs some action and if required, returns a report or result. This can range from a simple sensor checking if a product exceeds height restrictions to a baggage screening machine checking whether or not a product is secure. Some of the more complex objects are supplied by outside manufacturers and come with their own onboard software.

Each piece of luggage introduced into the system has a list of objectives to be completed, after which it has to arrive at a specific destination in a specific timeslot. For example, all products need to be scanned and screened for explosives and illegal content. This means that for each product there is a set of information that needs to be tracked and updated, while being accessible for information gathering by the LLC.

1.1.3 Compositional Interchange Format

The Compositional Interchange Format (CIF) is a hybrid automata based modeling language in development at the Systems Engineering Group of the Mechanical Engineering Department of the Eindhoven University of Technology [32]. The most recent version is CIF3 [9]. Using CIF, discrete event, timed and hybrid systems can be modeled. The main resource for CIF is [10] where information on the latest version, installation procedures, language tutorials and code examples can be found.

The CIF tooling is Java based and is integrated in the Eclipse Modeling Framework [12]. The CIF tooling allows for modeling requirements in the form of automata making it possible to built both a controller and a hardware model. Using SVG-based images these models can be tested in an interactive simulation, allowing the user to validate the software early on in the design process. The automata based structure of the requirements makes them very intuitive to understand and is less open to interpretation compared to textual requirements. Furthermore, once implemented, requirements are easily adapted by altering the automata.

The original goal of CIF was to provide a platform for transforming code from one language to another, acting as an intermediary. In recent development, CIF has become a programming language on its own, but the focus is still on generating code to different control platforms and languages. By modeling a control system in CIF it is possible to
generate code for multiple control platforms eliminating the need for manual recoding. Furthermore, if the transformation process to a control platform has been proven to be property preserving, the transformed code will adhere to the same requirements as the CIF specification. With the interactive visualization CIF offers, early testing of control software can be achieved.

1.1.4 Programmable Logic Controllers

Programmable Logic Controllers, or PLCs are small industrial computers used in industrial automation [24]. In the airport industry, PLCs are used almost exclusively for controlling the BHSs.

PLCs operate by running a cyclic program, illustrated in Figure 1.3. Every cycle, the PLC processes the input signals to compute the correct output signals for that cycle. The time a PLC spends on running one cycle is referred to as the cycle time. Generally, a lower cycle time is regarded as preferable, as during a cycle inputs and outputs are only read and written once. A long cycle time might cause the system to miss sensor signals or run actuators too long.

![PLC Scan Cycle](image)

Figure 1.3: PLC Scan Cycle [25]

While there exists an international standard for PLC programming languages, IEC61131-3 [15], each brand tends to use their own specific incompatible variation of this standard. In different regions of the world, different brands are often preferred. For an internationally
operating company like Vanderlande, this means that they often encounter different brands of PLCS in their projects.

PLC programs are built out of separate Program Organisation Units (POUs). There are several types of POUs and each POU contains a separate piece of code. In this project, the PLC programming language Siemens Structured Control Language (Siemens SCL) is used. SCL is a language developed by Siemens for control using their PLCS. Unlike ladder logic or structured text, which are older PLC programming languages, SCL is very similar to C in language structure. However, while the code may resemble C, it is still divided into POUs.

The most important POUs are the Organizations Blocks (OB), which can act as the main program or can be used for diagnostic purposes. Functions Blocks (FB) are used to create parameterized code capable of performing a specific function, which can be instantiated for multiple use. As each instantiation should be able to save data separately, each instance receives a Data Block (DB). A FB and each of its linked DBs form a separate piece of program, retaining values after use and running its code once each cycle. The fourth block is the Function, which uses input variables to perform internal calculations and give back an output, without retaining any variables of values. Functions can be executed multiple times each cycle, once for every call.

1.1.5 Code Generation

The code transformation from CIF to PLC used in this graduation project is being developed by Lennart Swartjes in his SUCCESS project. The goal is to create a code generation framework capable of transforming a CIF program to a specification in most major control platforms like PLCS or C/C++. The transformation can roughly be divided into two steps. These steps are a series of transformations to eliminate concepts not native in sequential programming languages and a final step to transform this code to the language of choice. For now, both the first step and the second step are still being extended in both the concepts used in the CIF code and the efficiency and readability of the resulting code. The choice of control languages is still limited, as expanding the transformable languages before a good basis is established is not beneficial.

Due to the large differences between CIF and PLC/SCL, certain restrictions are put upon the model by the transformation. Certain CIF concepts cannot be used or can only be applied in a specific way. Although the long-term goal is to eliminate such considerations, for now the target platform has to be kept in mind during modeling.
1.2 Preceding Work

This graduation project is the third consecutive project in this subject at Vanderlande and continues using the work of its two predecessors. In the first graduation project by Rik Kamphuis, a small loop of 8 conveyors was controlled with a PLC running code generated from a CIF specification \[18\]. The second project by Sjors Jansen focussed on extending this functionality to zone level and controlled a loop of 40 conveyors including sorting and merging functionality \[17\]. Due to the great amount of work that has already been accomplished and discussed in these projects, this report cannot always go in depth on the design process and decisions of the already existing model structure. For more information regarding these earlier projects, the reader is advised to look to the report of Sjors Jansen \[17\].

In the preceding projects, controllers were developed for all the basic sections used in standard flow, sortation and merging. In addition to the section controllers, zone controllers were created as well. Furthermore, hardware models of these sections and zones were created for use in the interactive visualization.

The work is also part of a larger collaboration between the Eindhoven University of Technology and Vanderlande, the SUCCESS project of PhD Student Lennart Swartjes. The SUCCESS project aims to build the modeling framework of code generation and transformations required to generate validated control software for multiple control platforms from a CIF source model.

1.3 Related Work

In a paper from 1987 \[27\], Supervisory Control Theory (SCT) was first proposed. By describing both unrestricted plant behaviour and the control requirements for a Discrete Event System (DES) with automata, a Supervisor can be synthesized. This Supervisor takes into account both the plant and the requirement automata to control the DES. In the years since, the research field of applying automata for control has become enormous and a complete description is far beyond the scope of this report. However, some superficial research was conducted to provide background and find existing results related to the modeling framework applied in this report.

For automata, an important distinction is made between discrete and hybrid automata. Discrete automata only use discrete variables to describe their state, altering variables is always executed in discrete steps. Hybrid automata combine both discrete and continuous elements for modeling automata \[2\]. A hybrid automaton contains a finite set of continuous variables described by a set of differential equations. While hybrid automata are more expressive and well suited for modeling real-time phenomena, verification of systems based on hybrid automata is very complex and can lead to state explosion even for relatively small systems. A special sub-class of hybrid automata are timed automata.
In timed automata, all continuous variables are clocks with a constant derivative of one [3].

Many tools have been developed for working with automata. Other than CIF, two well-known examples are Supremica and Uppaal [1, 19]. Uppaal is a system modeling tool based on timed automata. It allows for simulation and focuses on verification and validation of system properties. For verified properties, Uppaal guarantees that no series of transitions exist which can lead to a violation. However, when applying the verification tool, the size and functionality of the model is limited.

Supremica is a modeling tool based on finite-state untimed automata, offering synthesis and simulation [21]. Furthermore, it allows for the usage of definitions and instantiations of system modules to assemble large control systems [22]. However, modeling real-time processes without using time in any form is basically impossible.

Despite the broad scope of the research field, wide implementation of SCT in industry did not occur [14]. In [14], a number of problems with SCT were identified which created difficulties for implementations. Even though the controllers created in the SUCCESS project are not related to synthesis, most of these problems were also encountered in this project or previous projects. The following issues were relevant:

- Automata often provide multiple possible paths and are non-deterministic. In PLCs, all behaviour depends on the implementation and is deterministic.
- A translation from synchronous behaviour to equivalent non-synchronous behaviour in the implementation is required.

Research is also being performed on methods for defining formal control models for real-time systems. In [20], it is claimed that test cases with code generation showed that the resulting code contained less errors and was of a higher quality.

Much research focuses on application of SCT or automata-based control on PLCs despite these issues. Some academic and commercial packages support code generation to IEC61131-3 structured text. Examples are CIF and Matlab Simulink [6, 23]. Code generation to brand-specific programming languages such as Siemens SCL were not found.

In [28], a way to generate PLC programs from UML state diagrams is described. Finite or timed automata are represented in flow diagrams using UML. Code generation based on the UML specification results in a program of a PLC Ladder Diagram. The downside of such approaches is that they offer no way to include variables in the models. For instance, a boolean must be modeled as an automaton with two locations. Furthermore, UML or similar tools like XML, lack an environment for simulation or verification of the created controllers.

An approach using the BIP modeling language and Uppaal is outlined in [46]. BIP is a formal language using processes capable of describing real-time behaviour and incorporating variables [26]. The case study involves controlling a set of 8 gates performing...
complex movements using a rail system. Starting from a set of requirements, a BIP specification is modeled. Using a BIP2Uppaal transformation, the model is simulated and verified in the Uppaal environment. CIF also supports a transformation to Uppaal. However, Uppaal cannot handle verification of large models and only allows for a subset of continuous variables, clock variables with a constant derivative of one. Therefore, testing in Uppaal is not always beneficial and can be restrictive to the design process. For control models of the size developed in this project, it is not possible. This suggests that the framework applied in \cite{46} is not capable of controlling systems of the order of a Baggage Handling System. The next step described in \cite{46} is to generate and implement a PLC program. It is not mentioned to which standard or language type the code is transformed. The PLC program is implemented and tested on the gate system. The generated code is reported to be efficient and error-free. While the BIP language might be interesting for further research, the framework as proposed by \cite{46} seems incapable of controlling systems of the order of a BHS.

In addition to adjusting formal methods to the restrictions of the IEC61131 programming languages, there are also attempts to widen the scope and possibilities of PLC programming languages. IEC61499 \cite{16} was introduced in 2005 and should provide more possibilities to implement formally developed control models on PLCS. However, the standard is ambiguously worded, leading to differences between implementations. Furthermore, most implementations are academic, and industry is not adapting to the new standard \cite{33}. \cite{4,5} propose a mathematical definition and formal semantics for the standard which would eliminate the ambiguousness to achieve a more uniform implementation.

Although the research performed was very superficial, no other framework was found capable of generating controllers for large systems as achieved in this project. However, many different types of modeling frameworks and code transformations are in development and it is clear that many possible implementations for automata in control exist.

\section{1.4 Goal}

As a part of the SUCCESS project, this graduation work should provide the next step towards industrial application, controlling a small airport. In order to achieve this goal, a library of control definitions with the required functionality must be created. While the models created in the preceding graduation projects proved that CIF is a powerful tool in designing control systems, not all functionality and zones have been included. In order to control a small airport, the following requirements can be formulated.
The Check-In Desk, the Collector Zone, the Carrousel and the Identification Zone have not yet been modeled. Previous projects focused on material handling, sorting and merging. Zones related to input, output and measurement have not yet been included in the library.

In the current system, products have no identity or destination. In an actual BHS, each product has a specific goal and a list of security checks which has to be performed and documented. This product identity and the accompanying set of information need to be implemented.

As CIF is a language in active development, the language offers new functionality not available during the earlier projects. By implementing these new concepts and in general continuing to extend the functionality of the model definitions, the controllers can be improved.

In addition to these three points, the development of the CIF to PLC transformation has to be taken into account. The focus points are the modularity, readability and efficiency of the transformed code.

1.5 Outline

Chapter 2 uses some examples to explain the CIF concepts and terms used in this report. In Chapter 3, the implementation of a communication protocol and product identity is discussed. It handles the complete route of new system information from measurement to the update of the product information set. In Chapter 4, the ongoing development of model definitions and the considerations involved are explained using the standard Transport Section as an example. Chapter 5 describes the assembling of the control system of a small airport, and the real time implementation and testing on PLC. In Chapter 6, the conclusions and recommendations are given.

The main text of this report is intended to contain no confidential information. More information on certain confidential concepts and relevant Vanderlande sources is available in Appendix B.
Chapter 2

Modeling In CIF

This chapter gives some basic knowledge regarding CIF using a few examples. The goal is to illustrate the most important concepts in CIF and explain CIF jargon.

2.1 Automata and Events

2.1.1 Named Events

An automaton is a visible representation of the state of a discrete system. For instance, the motor of a conveyor has 2 discrete locations from a systems engineering perspective. It can be either On or Off. The assumption can be made that when the system is activated, the motor is always initially off. Automata move between locations using events or transitions. Listing 2.1 shows the automaton for the motor. Figure 2.1 gives a graphical representation of the automaton. The squares and arrows represent the locations and the transitions. The white square designates the initial location.

```cif
automaton motor:
    event e_on, e_off;

    location off:
        initial;
        edge e_on goto on;

    location on:
        edge e_off goto off;
end
```

Listing 2.1: Example Automaton motor
In the case of automaton motor, there are two events, $e_{on}$ and $e_{off}$. In the scope of control, the question is when an event is allowed to happen and how this can be influenced. The rule is, an event will happen, if it can happen. Preventing an event from occurring is called blocking the event and can be achieved by setting a condition for the event. Only when this condition is satisfied can the event occur. In Listing 2.2 and Figure 2.2, an automaton is added which uses the events declared in automaton motor. The purpose of this automaton is to link the motor behaviour to the status of the conveyor.

```cif
// Import the file with the motor automaton
import "model_cif_ex1.cif";

automaton status:
  event e_start, e_stop;

  location stopped:
    initial;
    edge e_start goto starting;

  location starting:
    // Synchronize with the event e_on
    // declared in motor
    edge motor.e_on goto running;

  location running:
    edge e_stop goto stopping;

  location stopping:
    // Synchronize with the event e_off
    // declared in motor
    edge motor.e_off goto stopped;

end
```

Listing 2.2: Example Automaton status
Because the events `motor.e_on` and `motor.e_off` are used in this automaton, these events are forced to synchronise between the two automata. Synchronisation means that both automata must participate in the event. If the event occurs, both automata must be in a location where the event is allowed. This means that the motor can turn on or off only when automaton `status` is in location `starting` or `stopping`. For instance, if the location `stopping` was the initial location of automaton `status`, both automata could not move to a different location. Automaton `motor` would need the event `motor.e_on` to occur to move to location `on` where event `motor.e_off` could occur, while automaton `status` would need event `motor.e_off` to happen before it could move to a location where event `motor.e_on` could occur.

### 2.1.2 Channels

A recent addition to CIF is a variation on the standard event, channels. Using channels, it becomes possible to transfer information in a transition. However, channels use some distinctly different rules.

A channel needs exactly one sender and one receiver before it can be initiated. When the channel is initiated, it transfers data of a type specified in the channel declaration from the sender to the receiver. Once the sender/receiver requirement is fulfilled, the channel acts much like a normal event would. Other processes can synchronise with a channel as if it were an event, but they will not receive the transferred information. In the same way a channel can be blocked, if a synchronizing automaton is in the wrong location or does not meet the requirements. Listing 2.3 and Figure 2.3 show a short example of the channel principle.
Chapter 2. Modeling In CIF

Listing 2.3: Application of channels

```
event int e_send;

automaton sender:
    disc int x = 1;
    location send:
        initial;
        // Sending the value of x
        edge e_send!x;
end

automaton receiver:
    disc int y;
    location receive:
        initial;
        // Receive a value and assign it to y
        edge e_send? do y := ?;
end

automaton synchronize:
    disc bool allow_send = true;
    location:
        initial;
        // e_send can only happen when
        // allow_send is true
        edge e_send when allow_send;
end
```

Figure 2.3: Automata overview of Listing 2.3
2.2 Variable Classes

There are three variable classes in CIF, discrete, algebraic and continuous. Of these classes, discrete and algebraic variables can have all possible data types, e.g. boolean, integer, real. Continuous variables are always of data type real.

Discrete variables are variables declared in automata which can be altered by updates in transitions. A discrete variable is a part of the state and has to be part of an automaton. An automaton which has a discrete boolean variable is extended in such a way that not just the location, but the value of the boolean is also required to fully describe the state of the automaton.

Just like discrete variables, continuous variables are a part of the state of an automaton. However, a continuous variable is always a real variable and can be assigned a derivative. The value of the variable is than dependent on the last assigned value, the time that has passed since that assignment and the derivative. Continuous variables are used extensively in this project to model conveyor displacement and measure the passing of time. The derivative of the variable does not have to be a constant value, it is also possible to make the derivative a changeable variable or even a conditional expression.

Algebraic variables are different from the other classes in that they are not a part of the state. As such they do not have to be declared in an automaton, but can be declared in any scope. Algebraic values offer a great amount of options. An algebraic variable is a function of your total state. It is possible to make an algebraic variable which is true when an automaton is in a certain location or which is equal to the sum of 3 discrete integers from three different automata. Algebraic variables are very useful for creating status and overview variables by combining relevant variables from different automata or processes.

Listing 2.4 and Figure 2.4 show some an example. The automaton distance measures how often a conveyor has run a distance equal to its own length. All three variable classes are used in different ways.
// Current speed of the conveyor
alg real speed = if motor.on: 1 // m/s
else 0 // m/s
end;

// Length of the conveyor
alg real length = 2.5; // m

automaton motor:
    event e_on, e_off;
    cont t = 0 der 1;

    location off:
        initial;
        edge e_on do t := 0
goto on;

    location on:
        edge e_off goto off;

    // Motor can only run for 10s at a time
    edge when t >= 10
goto off;
end

// Automaton measuring how often
// the conveyor has run its own length
automaton distance:
    // Current count
    disc int count = 0;
    // Current distance run
    cont dx = 0 der speed;

    location:
        initial;
        edge when dx > length
do count := count + 1, dx := 0;
end

Listing 2.4: Example of an automaton measuring how often a conveyor has run its own length
2.3 Definitions

When composing large systems, as in the case of this project, it occurs often that code is reused multiple times. For such purposes, CIF has automaton definitions and group definitions. An automaton definition is a parameterized version of an automaton which can be instantiated with different inputs.

Listing 2.5 and Figure 2.5 show a parameterized version of the automaton motor from Section 2.1. The automaton now has three inputs, the events to turn on and off, and an integer indicating the minimum time the motor must run before it can turn off. By instantiating this definition three times with specific events and values, distinctly different behaviour is accomplished for each automaton.

```cif
automaton def motor(event e_on, e_off; alg int max_time_on):
    cont t = 0 der 1;
    location off:
        initial;
        edge e_on do t := 0 goto on;
    location on:
        edge e_off when t >= max_time_on goto off;
end

// All motors start together
// Motor 1 is stopped independently
// Motor 2 and 3 are stopped together
// t >= max_time_on must be true in motor 2 and 3
// before event e_stop2 can occur!
event e_start, e_stop1, e_stop2;
motor1: motor(e_start, e_stop1, 10);
motor2: motor(e_start, e_stop2, 5);
motor3: motor(e_start, e_stop2, 10);
```

Listing 2.5: Instantiation of an automaton definition
A group definition is very similar to an automaton definition, but it concerns a group of cooperating automata instead of one solitary automaton. Using this principle, the model definitions in the previous projects were created. For instance, the model definition used to control a standard conveyor is a group definition containing several automata. There are automata handling tracking, PEC signals, conveyor status and calculation of the output signals. Because these automata are placed together in a parameterized group definition, any number of conveyors can be easily instantiated with unique parameters.

2.4 Interactive Visualization

CIF is also capable of several types of simulation. Of these types the most important is interactive visualization. Using SVG figures and elements, a visualization can be formed of the designed system. By interacting with this visualization via buttons and clickable elements, model behaviour and requirements can be tested.

The visualization is so useful because the properties of elements of the figure can be linked to a specific variable. For instance, it is possible to move elements, make them visible or invisible or change their size and color depending on variable values. Applications are for instance, products moving on a conveyor belt or a tank filling with water. Furthermore, elements can be linked to specific events. By clicking on this element, this event is triggered. This can be used to create a human-technology interface or start certain scenarios and control the simulation.

When working with a model definition, it is possible to define and link a group of SVG elements for that model definition. A single conveyor could for instance be represented by a standard group of elements, consisting of the conveyor belt, a sensor and some products. The visualization for one conveyor is shown in Figure 2.6. By changing text labels and the color of for instance the conveyor or the PEC sensor the status of the controller and hardware is visualized.
2.4. Interactive Visualization

Each instantiation of the conveyor model definition can copy this group of elements and alter properties such as conveyor length and PEC location, so a group of conveyors is created. By rotating and moving each conveyor to a specific spot, a network of conveyors can be assembled. Figure 2.7 shows the visualization of such a network of conveyors, forming a simplified version of a BHS test setup.

![Diagram of conveyor visualization](image)

**Figure 2.6: Visualization of a single conveyor**

**Figure 2.7: Interactive Visualization of a BHS**
Chapter 3

Data Communication

This chapter deals with the implementation of data communication. There are two major concepts to be implemented. Each subject is explained, concepts are posed and evaluated, and a final solution is presented. The first section deals with product information and identity. How should the product information be saved, accessed and altered? The second section deals with communication to and from lower-level parts of the system, specifically objects. For instance, data on measurements performed by equipment from outside controllers. This equipment is not an integrated part of the PLC network and has its own on-board control software.

3.1 Product Information

It is required that each product has a set of data that can be used to guide the product and to keep track of screening and routing processes. This data set has been prescribed by Vanderlande and is referred to in this report as a Baggage Record (BR). The following requirements can be formalized:

- Each product in data tracking should be uniquely linked to one BR in the PLC
- The BR should contain all the information of the data set prescribed by Vanderlande
- The BR is updated whenever new information or results become available in the PLC
- The BR is accessible or attainable by system controllers for information
3.1.1 Form of the Baggage Record

The prescribed set of information in each BR contains about 40 entries. Some entries containing separate sets of information themselves. In total it encompasses 102 separate items with the possibility of a variable going several subsets deep. As such, it is important that the record is ordered in a logical manner and that each variable can be easily retrieved. The simplest way to achieve this, is to use tuples with named fields. A tuple is a custom-made variable type consisting of several sub-fields. Each sub-field has a name in the type. That sub-field can be found if both the declared name of the tuple variable and the name of sub-field in the type are known. As a sub-field of a tuple can be a tuple in itself, the complexity of the BR can adequately be captured using a large tuple type. An example of a tuple type declaration and use is shown in Listing 3.1.

```plaintext
type destination = tuple(int number; real class);

type br = tuple(bool used; destination dest1, dest2);

automaton A:
    event e_product;
    disc br br1 = (true, (1, 10.0),(-1, -1.0));

    location A1:
        edge e_product do br1[dest1][number] := 2;
end
```

Listing 3.1: Example of declaration and usage of Tuple Types

In the example a tuple type is defined containing an integer and another tuple type. A variable of this tuple type is declared and using an event its value can be altered. For the tuple type defined for the BR, look to Appendix C.

There are two major concepts to store the BR for each product, a global database or moving the BR along with the product tracking information through the TRS controllers. As the data communication needs to be implemented in CIF and transformed to Siemens PLC, restrictions in both languages are considered. This is elaborated on in the next section.

3.1.2 Location of Product Information

Considerations for CIF and PLC

In storing and accessing variables, parallel processing languages like CIF differ from the more commonly known programming languages like C or Java. While it is not impossible to use globally accessible and alterable variables (Global Write/Global Read) in parallel processing, it is known to be a complex issue often leading to errors. As multiple processes
3.1. Product Information

can simultaneously alter the variable, issues arise if for instance two processes in one transition try to change the variable. All updates in a transition are simultaneous, so which process has precedence and decides the new value? Furthermore, validation and checking for correctness become far more complicated. For these reasons, a Global Read/Local Write property is used in CIF. This means that CIF has no global variables in the traditional sense. This property specifies the permissions of writing and reading variables between automata. If automaton A possesses variable A1 than automaton B can read A1 but cannot change the value of A1. Changes to A1 can only be carried out by automaton A. This way, the state of every automaton can only be changed by that automaton itself. Furthermore, it forces the user to implement structure by deciding which automaton has ownership over which variable.

This property naturally influences the choice of where and how to store the BRs, as it needs to work in both CIF and on the PLC. A central automaton as a BR database would be easily accessible from the entire system. However, updates from the system could only be carried out indirectly, through events or channels. An alternative is moving each BR along with the position information in the TRS controllers. This concept would make it much easier for a TRS controller or a local process to carry out an update, but the dynamic data location makes it much more difficult to access a BR or carry out an update from a non-local process.

In IEC 61131 PLCs have no restrictions in reading/writing rights. However, in PLC code, memory has to be allocated for every variable on initialization. In both concepts, memory has to be reserved for every BR that might be used. The concept which requires the least BR variables would be the most efficient. It should be kept in mind that the final implementation should be memory and thus variable efficient.

In both software environments, the number of computations required for the concept of local storing is larger. Every time a product moves to another conveyor, the BR needs to be transferred as well. In a global database, the BR would not be moved between initialization and deletion, saving on computation time.

Origin of Updates

In the BHS, updates are only initiated from within the PLC; HLC never update a BR directly. Updates originate in almost all cases from measuring objects, equipment supplied by outside manufacturers e.g. volume scanners and label scanners. These updates concern only one or a few related object properties. However, measuring, processing and reporting the acquired information is not instantaneous, causing a time delay. Products generally do not stop during measurements, but are measured while moving. It is possible that products move to downstream conveyors before the update can be initiated, as visualized in Figure 3.1. The main argument for local storage was direct access to the BR, which is not true in the case of measurement delays. For local storage, an indirect way of updating would be required, just as with the global database. Indirect access could be achieved by sending update information ahead to the decision point, waiting for the product and
so on. However, these methods are not very robust and for every update a number of unnecessary checks need to be performed to find the right product. Furthermore, it fragments the information of the BR throughout the system which defeats the purpose of having one set of information per product. As a dynamic data location offers no advantages, a global database is chosen.

In the next section, concepts for this database are discussed as well as the best way to update a BR while not violating the Local Write/Global Read property.

### 3.1.3 Form of the Database

Depending on the function of the area and the combined capacity of the conveyors, each PLC should be capable of storing a few hundred BRs. In the previous section, the choice was made to use a globally accessible database. In this section, the form of this database is discussed.

**Array of Tuples**

In CIF, the best data construction to store all the BRs is an array. Arrays are lists of predetermined and invariable length that are native to both CIF and PLC. Each spot in an array is accessible using indices. The index of the spot of each BR in the array could be used to link it to a specific product. The internal PLC product ID would be the array index in which the corresponding BR is stored. This array is henceforth referred to as the BR Array. This array can be handled as a single variable and can be stored in a single automaton. Except for the handling of updates, the database has been defined. Listing 3.2 and Figure 3.2 show the declaration and use of an array in combination with adding and removing a BR.
3.1. Product Information

// Tuple type containing the BR
type br = tuple(bool used; real br_values);

automaton BR_Array:
// Channels for accepting and deleting a BR
event br e_new_br;
event int e_delete_br;
disc br empty_br = (false, 0); // Empty BR
disc list[3] br array = // BR Array
    [(false, 0), (false, 1), (false, 2)];
// free index refers to a free index in the array
disc int free_index;

location A1:
    edge e_new_br? do array[free_index] := ?;

    edge e_delete_br? do array[?] := empty_br;
end

Listing 3.2: Example of declaration and use of the Array Type

Figure 3.2: Automaton overview of Listing 3.2
Placement of New Br Records

In order to operate for an extended amount of time, the BR Array should efficiently make use of all available indices of the array. It is important that a strategy for placing new BRs is implemented. The choice was made to store new BRs on the lowest free index. A small algorithm was devised to accomplish this goal. A variable indicates the lowest free index, which is initiated at 0. If a product is registered, it receives index 0 and a function is called seeking the next free index in the array. In this case, that is index 1. If the product on index 0 is then deregistered, the BR Array checks if this newly freed index is lower than the known lowest index. If this is the case, such as in this scenario, the variable is overwritten with the new value. In this way, a new BR is always placed on the lowest available index, keeping the first parts of the array automatically most densely “populated”.

Updating BR Records

Updates cannot be performed directly. An indirect method has to be used, transferring information from one automaton to another. The most straightforward way to transfer data between automata is channels. Channels are events which carry out point-to-point communication requiring both a sender and receiver. As long as at least one sender and a receiver are available, the channel can send information. Not all updates have the same format, several types can be distinguished. Each updating object measures different criteria so each object updates different variables. As such, there are at least as many types of updates as there are measuring objects. By reviewing the Vanderlande system, it can be assumed that the number of update types could be restricted to less than 10.

Each update type could use its own channel, or only a single channel could be used for every update. However, when altering the value of a tuple on a transition, CIF has some restrictions regarding the field name. When altering a tuple, CIF forces the user to statically set the updated subvariables. It is not allowed to dynamically set the subvariables to be updated, based on for instance the information received with the channel. As such, one channel cannot be used to carry out an update for each update type on a single transition, but at least two transitions are required to carry out a single update. This can be prevented by assigning an update channel to each update type. This way it is known when declaring the event which tuple fields are affected and they can be statically programmed. For this reason, multiple update channels are introduced. One for each update type, so updates can never be wrongly assigned. An example is shown in Listing 3.3 and Figure 3.3.
automaton update_label_nr:

    location:
    initial;
    // Sending edge
    // Updating the label number of product 37
    edge BR_array.e_update_label_nr!(37, 84829192);
end

automaton update_weight:

    location:
    initial;
    // Sending edge
    // Updating the weight of product 102
    edge BR_array.e_update_weight!(102, 9.2);
end

automaton BR_array:

    // Declaring the BR type
    type BR = tuple(int label; real weight);
    // Channel declarations
    event tuple(int product_id, label_nr) e_update_label_nr;
    event tuple(int product_id; real weight) e_update_weight;
    // BR Array of length 200 and type BR
    disc list[200] BR array;

    location:
    initial;
    // Receiving edges
    // Using the variable product_id any product can be updated
    edge e_update_label_nr?
        do array[?][product_id][label] := ?[label_nr];
    edge e_update_weight?
        do array[?][product_id][weight] := ?[weight];
end
3.1.4 Product Identity

In the previous section, the final concept uses an integer pointing to the right index in the BR Array. Implementing this product ID would give each registered product a unique identity. In the controllers created in the previous projects, there was no need for information to pass along with a product, as products were anonymous. The head and tail data were not explicitly linked, whenever a head or a tail crossed to the next conveyor a synchronising event would erase the marker in the upstream conveyor and create a new one in the downstream conveyor. As this was modeled before the introduction of channels there was also no straightforward alternative to transfer product tracking information.

In order to send along a product identity, the procedure of transferring a product has to be replaced or extended with a channel. This also allows for a review of the current method of storing a product. Now that information can be easily transferred, the head and tail coordinates can be sent along with the product ID. This would allow for a single tuple variable containing all the information concerning a product, in addition to having only one container for all the products instead of having 2 or 3 for head, tail and product ID. Bundling product information prevents loose heads and tails in tracking and all the information concerning a product is always stored on only one conveyor. Implementing these concepts requires the algorithm used to transfer products to be redesigned. As the Transport Section is subject to a number of different changes and proposals, these are discussed all at once in Chapter 4.
3.2 Object Communication

This section deals with communication to and from objects. As written in Subsection 1.1.2, objects are separate parts of the system which can receive requests and return results. In some cases these objects come from external suppliers and their operations are controlled by on-board software. Most measurements performed on products in the system are carried out by objects. As discussed in Subsection 3.1.3 these objects are responsible for the updates to the BR. The solution proposed in Subsection 3.1.3 to perform these updates is a channel relaying the information to the BR Array where the update is handled internally. As the equipment comes from an outside supplier it is not possible to simply alter the on-board software. In the next section this problem is addressed and the CIF implementation is shown, using the example of a new zone communicating with an updating object.

3.2.1 Identification Zone

The Identification Zone (IDZ) scans the labels of the products, providing a way to link a product in the PLC to the global High Level database. It consists of two conveyors where products are scanned while passing through the zone. There are several variants where additional objects also measure weight and volume, but in this example only the label scanner is considered. An overview of the IDZ is shown in Figure 3.4.

![Figure 3.4: Overview of the IDZ](image)

The label scanner has an on-board controller, so a process in the CIF control models needs to interact with these on-board controllers. By looking into the IDZ, the following specification was found for coordinating a measurement with the label scanner.

1. When a product comes in range, a trigger is sent to the object by a boolean signal
2. The scanner start a measurement
3. When the trigger ends the measurement is stopped
4. The scanner reviews the measurement and creates a measurement report introducing a small delay
5. The object relays a message containing the results back to the PLC
6. The object quickly switches a bit on and off to signal that a message was sent.

The trigger described can be both software or hardware controlled. For instance, based on tracking information or by a PEC sensor readout. The measurement report contains up to three possible readings ordered by read count. Therefore, the communication between the PLC and the object consist of the trigger bit as output, the new message bit and a number of bytes reserved for messages as input.

While it is possible to integrate the interfacing process into the zone controller, the choice was made to create a separate definition for the interfacing process. In this way one zone controller can be used in combination with object interface definitions for function variants, and objects can be used regardless of a specific zone. This choice led to the data structure of Figure 3.5.

The interfacing process consists of an automaton with three locations. It can be On or Off depending on the zone controller. In the location On it can detect a new message through the new message bit and move to the third location, Message_Handling. In this location the incoming Label Code is compared to the one currently known. If it is a new code it overwrites the old one, else if it is a no-read or the same Code the BR is not changed. After the message has been processed, the automaton returns to the location On and awaits a new message. The trigger bit is handled in the location On using tracking information from the first conveyor.

In conclusion, the general solution for objects is to create a separate controller which interfaces with the object. Additionally, this controller uses the update channel for that object type to relay updates to the BR Array. As the manufacturer and product type of objects differ from project to project, the controller is also often project specific. As such controllers were not created for all object types, but the update channels were defined in the BR Array. With the general method shown in this section and the object update channel, a controller can be custom made for each project and each unique object interface.
Chapter 4

Scaling, Interfacing and Reuse

By instantiating the created model definitions multiple times, using different parameters, it is easy to assemble large control systems. However, whenever a non-standard situation occurs, the standard model definition might not be usable. In this chapter these model definitions are re-evaluated with the goal of introducing scalability, improving interfacing between definitions and to find a balance between having versatile but inefficient definitions or using many different definitions. Unless mentioned otherwise, all concepts shown are implemented and used in the real time implementation described in Chapter 5. Using the TRS, the general way such issues were handled is explained. The TRS is the most intensively used model definition, and has many different possible configurations, features and requirements.

4.1 Scalability

In the Transport Section (TRS) definition developed in the previous projects, all the required functionality was included. However, it was very rigidly modeled requiring programming changes for even small alterations. For instance, conveyor length was a modeling parameter, but the amount of products that could be kept in tracking was not. The model definition held a place for one PEC signal and tracking was modeled around the concept of separating products before and after the PEC sensor. As conveyor size can range from slightly longer than a meter to a hundred meters with multiple PEC sensors, this definition is insufficient. In this section, the tracking and PEC input handling are reviewed and remodeled to make them more scalable.

4.1.1 Scalable Tracking Size

It is important that each conveyor has room in tracking for a sufficient amount of products, while it also preferable that smaller conveyors do not waste memory with overcapacity.
Originally, tracking was modeled as a series of continuous variables, half of which heads and the other half tails. Sending a head or a tail to the downstream conveyor required moving all tracking information up to the next variable and overwriting the one in front. When using an in- or outfeeds, this created additional difficulties. The sending and receiving algorithm could not place or remove a product when it was not the first or last product on the conveyor. An example of receiving a head from the upstream conveyor is shown in Listing 4.1.

```plaintext
// This edge adds an additional head to tracking
// h1 to h4 are continuous variables representing a head
// integer hn represents the number of heads already in tracking

event ch_acc do
    if hn = 0: h1 := 0.0, hn := 1
    elif hn = 1: h2 := 0.0, hn := 2
    elif hn = 2: h3 := 0.0, hn := 3
    elif hn = 3: h4 := 0.0, hn := 4
end;
```

Listing 4.1: Non-scalable solution applied in previous projects

Continuous variables are a very simple way to model a moving object, as explained in Chapter 2. However, continuous variables cannot be placed in containers such as tuples or arrays leading to constructions that do not easily scale, as shown in the example above. In order to introduce scalability into the tracking, a way has to be found to store all the tracking information in a single container. For instance, using only one continuous variable representing the displacement of the belt. By introducing an array of reals, the position of all products could be described using this array and the displacement variable.

As discussed in Subsection 3.1.4, the introduction of channels and the need to transfer information allows for storing product information in a new way. Instead of having separate tails and heads and synchronizing with the downstream conveyor when one transfers, it is possible to send tracking information along using a channel. Bundling all product information in one place allows for sending it at the same time, ensuring that the information is always on the same conveyor. All product information can be stored in a tuple such as “type prd = tuple(real hd, tl; int br_id)”. By taking an array of type prd instead of real, the 10 continuous variables of the previous version can be reduced to a continuous variable measuring the displacement and an array holding tuples with each tuple representing a product. By adjusting the length of the array, each conveyor can hold a scalable amount of products.

Some choices and conventions need to be set for this solution to be implementable. For instance, the order of products in the array. A suitable order is placing the first product in index 0 and the n\textsuperscript{th} product in the n-1\textsuperscript{th} index. This way, it is always known which product is first. By keeping track of the value of n, the tracking automaton only needs to check the position of the product in the index 0. New products coming in from the downstream conveyor can be placed in position n. Variable n would indicate both the number of products on the belt and the next free index in the array. A downside of this
4.1. Scalability

order is that whenever a product is transferred, removed or merged in, products must be moved to restore the right order. By using an array, this can be done in a function using a while loop. As this means the removal or inclusion of a new product, this is the ideal time to update the array with the displacement variable using a third function. An overview of the implemented array and the important variables is shown in Figure 4.1.

An example of a function adding a product to the array is shown in Listing 4.2. A figure showing the array and the movement of products is shown.

```
[ (hd0, tl0, id0), (hd1, tl1, id1), (hd2, tl2, id2), (hd3, tl3, id3), (-1,-1,-1), (-1,-1,-1) ] + dx
```

Figure 4.1: Scalable solution via a Tracking Array

```
type prd = tuple(real hd, tl; int br_id);
// Function to add a merging prd
func list[6] prd add_prd(list[6] prd tracker; int n; prd new; real dx) :
    int i = n-1;
    // Update the list with dx
    tracker := updatelst(tracker, n, dx);
    // Start from the back
    // While the new product is not in place
    // Keep moving it up one spot
    // And move the other products down one place
    while tracker[i][hd] < new[hd] and 0 <= i:
        tracker[i+1] := tracker[i];
        i := i - 1;
    end
    tracker[i+1] := new;
    return tracker;
end
```

Listing 4.2: Function adding a product to the Tracking Array

The sending algorithm used in previous projects is no longer valid when transferring all information regarding one product at once. A new protocol was developed which is shown in Figure 4.2. The boolean variable Transfer is introduced and indicates whether a product is in transfer between the two belts. In the figure, belt 1 sets the value of Transfer and exports it to belt 2. If Transfer is true, both belts are aware of a transfer and can run the motor while they may not have a product in tracking.
1. A product approaches the end of the conveyor, while still being outside of the PEC range.

2. Once the product is in PEC range, the head position is updated in tracking.

3. The head of the product leaves the first section. To notify belt 2 that a product is on the belt of which no data is known in tracking, the boolean Transfer is set to true in belt 1.

4. The product is exactly halfway between the two sections, and the product data is transferred. The assumption is made that the conveyor on which the largest portion of the products rests, controls the product speed. In order for belt 1 to know when the Transfer signal is no longer required, a countdown is started linked to the belt displacement counting down from half the product length.

5. The tail position of the product is determined by the PEC sensor, while the data is already on the second section. Although this only occurs if the distance between the end of the belt and the PEC sensor is smaller than half the length of the product, a solution is required. An extra channel was introduced as is explained in Subsection 4.1.2. Normally, the tail position is detected before the product data is moved and the tracking position is updated locally. In this example, this would be between step 2 and 3.

6. The product has completely cleared belt 1 and the Transfer signal is set to false as the countdown becomes zero.

4.1.2 Multiple PEC Sensors

In the previous models only one PEC sensor was included, which was interwoven with the tracking. In tracking, the distinction was made between products before the PEC and products after the PEC. Whenever a product was detected, the first product in tracking before the PEC was checked. If this was the product detected, it was moved to the secondary tracking. If this was not the case, a UFO was introduced to the secondary tracking. When considering only one sensor, this is a very logical method. It neatly separates the next product to be detected by the PEC and the next product to transfer to the next conveyor. In Subsection 4.1.1 the new tracking was outlined which allows for a different way of handling PEC inputs, so that any number of PECs can be included.

When introducing multiple PEC inputs, it is no longer possible to introduce a before and after PEC distinction. In the case of two PECs, this would have to be fragmented further to three regions, before the first PEC, between the two PECs and after the last PEC. For three PECs or more, it would be different again, making it hard to scale. Introducing the distinction separately for each PEC would require multiple versions of the tracking array, one for each PEC. A simpler method is checking for each PEC if there is a product expected in their range. If there is product expected and a head is determined, it can
4.1. Scalability

Figure 4.2: Transfer Protocol
be linked to that product. If a head is determined while there is no product in range, it must be a UFO. This way no distinction is made and all products are in the same tracking array. Adding a PEC is matter of copying this automaton and changing the parameters to check in a range around this new PEC sensor.

To implement this concept, the set of automata controlling the PEC in the old tracking method need to be adjusted and turned into a model definition in itself. This definition could be instantiated within the TRS model definition for every PEC. There are three automata in this PEC definition, of which the important one here is an automaton handling synchronisation with tracking and detection of products. An algebraic variable is introduced which checks, using the tracking array, if there is a head or a tail in a certain range. The automaton can use this variable to determine when to expect a product and the index of this product in the array. Whenever a head or a tail has been determined, the location of the PEC and the index of the product are sent to the tracking automaton using a channel. In case of a UFO, the PEC location is sent using a channel and the tracking automaton can add a product. If a product was expected but never confirmed, a missing report consisting of the product index in the array is sent to tracking. The function used, checks for products in range until it either finds one or it encounters a product which has yet to arrive at the start of the detection range.

As illustrated in Subsection 4.1.1, a last issue occurs if a PEC is located very close to the end of the belt and a longer product is in transfer. It is possible that the tail of this product is only detected once the product information has already been sent to the next conveyor. This is a new problem as in the previous tracking concept this was not possible. It is solved by introducing a variable detecting whether a determined tail was caused by a product in transfer. If this is the case, the PEC automaton sends its location as seen
from the downstream conveyor to the tracking automaton of the downstream conveyor. There, the tail of the last product in tracking can be updated using the adjusted PEC location. An overview of the four different channels from PEC to tracking is given in Figure 4.3.

4.2 Interfacing

Interfacing between model definitions is especially important for the transport section. As the transport section is used both in relatively simple and complex situations. It is important that the Transport Section has a versatile interface, while avoiding having too many dummy parameters and mostly unused features. An example illustrating a possible conveyor configuration featuring an infeed and outfeed is shown in Figure 4.4.

4.2.1 Multiple In- and Outfeeds

The standard interface with up- and downstream conveyors consists of a set of variables and events/channels going to the downstream conveyor and a set of variables going upstream. The downstream conveyor relays back if it is ready to receive a product and whether or not is in a state which the conveyor needs to follow. The upstream conveyor exports channels used for sending products and sends tracking information on the upcoming product. Furthermore, it exports a Transfer signal and whether or not it is in a state which the conveyor needs to follow. An overview of the exchanged variables is shown in Figure 4.5. The four Errorstates in the Figure are unique states which all must be responded to with a different reaction.
Figure 4.5: Overview of the input variables for the standard TRS definition from the up- and downstream conveyors

Adding one in- or outfeed to this interface has been accomplished. An infeed to a conveyor can receive the same inputs as the standard downstream conveyor and an outfeed usually receives specific variables adjusted by the zone controller. Furthermore, there are special infeed and outfeed channels sending product information going to and from each conveyor. In case these are not used, dummy channels are given. However, using more than one in- or outfeed causes technical issues in CIF over channel/event ownership and non-deterministic choice of receiver. In this section these issues are addressed for both in- and outfeeds.

**Infeeds**

In order to keep the interface simple and easily scalable, it is preferred that infeeds use only one channel to send product information, regardless of the amount of infeeds. Otherwise a custom definition is required for each possible number of infeeds connected to one conveyor. Multiple infeeds can work with one channel, each infeed can send information over the channel to the only receiver, the downstream conveyor. However, when using only one channel two considerations must be made:

1. The receiving conveyor cannot identify the sender by the channel used, so the source location of the product is unknown
2. Each sending conveyor exports its own channel, so in this case there are as many input channels as infeeds

The first issue is a result of having multiple possible senders. If the conveyor cannot identify the sender, it cannot know at which infeed the new coordinates need to be placed. This was solved by including as an input for every conveyor, the junction location with the receiving conveyor. In the case of a standard interface or an outfeed this is zero. For an infeed it is where it connects with the receiving conveyor. The sending party adjusts the coordinates of a product being transferred to its junction location with the receiving conveyor. The receiving conveyor only has to place the coordinates in the correct spot in the tracking array. Without knowing from which infeed the product originated, it can still be correctly placed. The head coordinate of a product sent from an infeed to a downstream conveyor is then “head coordinate – conveyor length + junction coordinate”.
The second issue has to do with channel ownership. In the standard interface, each conveyor locally declares the channel by which it will send product information. This channel is an input parameter for the downstream conveyor. Regardless of the amount of infeeds, the local conveyor still has only one input for a merging channel. Meanwhile, each interface is sending its product information by a different one, as illustrated in Figure 4.6. As there is no way to merge multiple channels into one, an alternative must be introduced. Preferably without having to create two-step transitions in sending products.

Creating an interfacing automaton receiving information from all the separate channels and relaying it forward to the receiving conveyor would not be complicated. However, it would disrupt the atomic nature of the transition. Furthermore, by its very nature it could not be standardized in a scalable model definition, as the number of inputs would vary. An example for a situation with three infeeds is shown in Listing 4.3 and Figure 4.7.

```plaintext
// Figure 4.6: Illustration of the Channel Ownership Mismatch

Listing 4.3: Interfacing automaton for resolving channel ownership
```
The problem is that ownership of the channel is in a definition. If the definitions were to be changed to owning the channel by which they receive products and importing the channel by which they send, a similar problem would be created for a situation with multiple outfeeds. In the case of  \( n \) outfeeds there would be \( n \) channels receiving products while the TRS definition only expects one channel. So in order to solve the problem for all configurations, channels sending product information cannot be locally owned. They must be declared at the same level as the model instantiations. By implementing this solution, every TRS definition would receive two extra inputs, the channels sending products which are now locally declared. As both the sending party and the receiving party expect one channel as an input, it is easy to declare one channel and give it as input to all sending and receiving parties. However, for every connection between two conveyors, an event must be declared independently on the highest scope. An example for declaring a loop of 4 conveyors is shown in Listing 4.4.

```python
automaton def trs(event real e_acc, e_send):
    // Controlling the TRS location;
end

event real e_send_1_to_2, e_send_2_to_3, e_send_3_to_4, e_send_4_to_1;

TRS1: trs(e_send_4_to_1, e_send_1_to_2);
TRS2: trs(e_send_1_to_2, e_send_2_to_3);
TRS3: trs(e_send_2_to_3, e_send_3_to_4);
TRS4: trs(e_send_3_to_4, e_send_4_to_1);
```

Listing 4.4: Example of instantiation for a loop of conveyors

In conclusion, to allow for multiple infeeds to one conveyor, two changes had to be made to the interface between conveyors. Firstly, conveyors now send the coordinates of the product as seen from the receiving conveyor, so in the case of an infeed this could be
4.2. Interfacing

halfway the conveyor. Secondly, sending or receiving channel ownership was moved to the same level as the model instantiations and all these channels are inputs. This restricts the need for extra model definitions, while making sure that channels ownership does not cause problems in conveyor configurations. While this alteration was required to create scalable solutions, it does mean extra time during instantiation of conveyors and linking conveyors becomes more complex. However, automatic generation of CIF controllers would eliminate this difficulty.

Conceptual Solution for Outfeed Modeling and Control

The problem of channel ownership with multiple infeeds was solved while making sure that the same problem did not occur for one conveyor having multiple outfeeds. However, with multiple outfeeds, the theoretical basis of channels has to be considered. In this project, a situation with multiple outfeeds was not encountered. However, a possible solution was developed and is described in this section.

A channel can only send information if there is both an available sender and receiver. In the case that there are multiple available receivers, the sender will choose one of the receivers non-deterministically. The issue created is that the sender cannot choose to which receiver the information will go. In the case of multiple outfeeds this means that the receivers, the outfeed conveyors, should only be able to receive when a product is being sorted out to them, restricting the number of options the sender can choose to one. Each outfeed could take tracking information from the sending conveyor and the status of sorting equipment into consideration to determine if it should be eligible to receive. However, in the case that two products are both being sorted out simultaneously, two eligible receivers still remain. In this case, one outfeed could receive product information on a product being sorted to another outfeed. A situation with two eligible receivers is shown in Figure 4.8.

![Figure 4.8: Situation with two eligible receivers](image)

In order to be sure that the product information being sent is meant for them, outfeeds would have to able to review the information being transferred before accepting to
receive. A way of accomplishing this in CIF is to make use of invariants. Invariants are boolean statements declared by the user, which should always be satisfied. For instance, if there was an event incrementing a variable x and and invariant stating that x should be smaller than 10, this event would be blocked once x becomes 9. In the same manner it would be possible to give each outfeed an id number. When sending a product, an id number is sent along indicating the outfeed the product should go to. Each outfeed would have an invariant, stating that the outfeed number received from the channel should be equal to its own outfeed id. The invariant works by looking forward into the next state as if a certain event were to take place, if this new state is not allowed by the invariant, it blocks the event. Any product sent with an outfeed number different from the outfeeds own number, would not be allowed. This way the sending channel would only have one eligible receiver and the receiving channels can only receive the right product. A possible implementation is shown in Listing 4.5.

```cif
// outfeed_id indicates the id of the outfeed
// to which the product is sent
type prd = tuple(real hd, tl; int br_id);
event tuple(prd prd; int targeted_outfeed_id) e_send2outfeed;

automaton outfeed:
    // The id number of the outfeed
    const int outfeed_id = 3;
    // Integer for saving the outfeed id
    // received with e_send2outfeed
    disc int rec_outfeed_id = 3;

    // The invariant stating that the
    // received outfeed id should always
    // match the id of the outfeed
    invariant (outfeed_id = rec_outfeed_id);

    location:
        initial;
        // Edge updates tracking with the new product
        // and updates rec_outfeed_id
        edge e_send2outfeed? do // Add product to tracking
            rec_outfeed_id := ?[targeted_outfeed_id];

Listing 4.5: Possible implementation of the invariant concept
```

While invariants are a standard part of CIF, they are not yet part of the supported functions in the transformation. Stepping back to the previous state is possible in the CIF simulator, but is an unknown concept in most control platforms. On the other hand, the invariants are a solution to a unique CIF problem, the non-deterministic choice. In PLC or other platforms the invariant might for instance be part of the guard, as using if statements a deterministic choice of receivers can be forced. The issue in this section was
4.2. Interfacing

caused by preferring to use only one channel while still having multiple receivers. An alternate solution or 'quick fix' is giving each outfeed its event, at the cost of scalability and re-use of the definition.

4.2.2 Window Inshooting

Efficiently controlling long line of infeeds is a complex problem. If all infeeds can deposit products at will, infeeds at the start of the conveyor have a much shorter waiting time than infeeds at the end. In order to conquer this complexity, there are several strategies or algorithms for reserving space or finding gaps. Preferably in a way that all infeeds can offload their products with equal waiting time. An example of such a situation are the check-in desks at airports. Depending on the size of the airport, dozens of check-ins can be used in parallel, all depositing products on the same series of conveyors. There is a zone handling all these check-in infeeds called the Collector Zone and the solution implemented in this project is a window algorithm. Vanderlande employs an elaborate window algorithm focussed on ensuring maximum efficiency of conveyor space. A simplified algorithm is used in this project as a proof of concept.

Using windows, a window generator is placed at the start of the zone. A window is free space reserved for a specific infeed on which a product can be placed. When an infeed is ready to deposit a product, a request for a window is sent to the generator. The generator places a window on the conveyor. As the conveyor moves, the window moves downstream to the relevant infeed. When the window is alongside the infeed, the product is deposited in the window. An example of a Collector Zone with three infeeds using a window algorithm is shown in Figure 4.9.

![Window Infeed 3
Infeed 1 Infeed 2 Infeed 3
Window Infeed 1
Infeed 2
Window Infeed 2
Flow

Figure 4.9: Example of a Collector Zone with infeeds waiting for a window

With windows, every infeed has equal opportunity to deposit it products. While many algorithms have been developed for assigning and handling windows, due to time constraints a very simple one is implemented here. Each infeed receives a number, they use this number as their identity to request a window. When a window with the same ID as the infeed approaches, the infeed deposits the product. The zone handles the window
generator. It keeps a chronological waitlist of all window requests. When there are waiting requests and there is space for a new window, one is generated for the first in queue. Windows are modeled as virtual products, a tuple containing two reals indicating the head and the tail and an integer for the infeed ID. Similarly as in Subsection 4.1.1 an array and a displacement variable are used. This displacement variable is linked to the conveyors of the zone. The window array tracks a window from the generator over all conveyors in the zone. Whenever a window leaves the last conveyor of the zone, it is deleted.

For infeeds to deposit product to windows, a different interface is required. Infeeds do not have to check if they can send products collision-free, as the presence of a window with the right ID should guarantee that. However, infeeds need access to the window tracking and displacement variable to find windows in range with the right ID. The infeed waiting for the reserved window is very similar to the situation of a PEC expecting a product. A special Check-in Conveyor model was created to reflect the need for a new interface which is elaborated on in Chapter 5. The automata adding and updating the window positions are quite complex and cannot be shown here. Listing 4.6 and Figure 4.10 show a simplified example of the interfacing between an infeed and the window generator.

```plaintext
event int e_req_window;
event e_create_window;
// Boolean indicating whether
// a new window can be added
alg bool window_free = true;

automaton req_window:
    disc int infeed_id = 1;
    location:
        initial;
        edge e_req_window!infeed_id;
end

automaton gen_window:
    disc int n_q = 0; // Number of requests
    disc int n  = 0; // Number of windows
    location:
        initial;
        edge e_req_window? // Add to queue!
            do n_q := n_q + 1;
        edge e_create_window // Add window
            when window_free and n_q > 0
                do n := n + 1,
                    n_q := n_q - 1;
end
```

Listing 4.6: Interface between an infeed and the window generator
4.3. Reuse

Even though the TRS already encompasses most of the required functionality, there are some features which have not yet been implemented, such as bi-directionality or master-slave configurations. These kind of features demand a large change to the model definition, in the case of bi-directionality the entire interface with other conveyors has to work in both directions. If this was implemented into the basic definition, every conveyor without a bi-directional drive would have a large number of unused inputs. On the other hand, every feature for which a separate controller is created, creates the need for more separate controllers. For instance, there are conveyors which are bi-directional and which function in a master-slave configuration. Deciding to make separate controller definitions for both of these features would create a necessity for a third definition, combining the two features. In this section, implementation of these features and the consequences for the TRS definition are discussed.

4.3.1 Bi-Directional Conveyors

Bi-directional conveyors are, as the name implies, conveyors which can move in both directions. This does not have a large influence on controlling the hardware, as instead of a go and stop signal, the go signal is now split into forward or backward. However, conveyors running in only one direction always interface with same conveyors in the same way. If bi-directionality is introduced, both bordering conveyors can be upstream or downstream, depending on the direction of the drive.
In- and Output Variables

Bi-directional conveyors have a boolean input prescribing in which direction they should be oriented, forward (fw) or backward (bw). The first thing that was added was an automaton keeping track of this input. Whenever it switches, an event is triggered, so that the switch can be carried out simultaneously in all automata of the model definition. The motor control was adjusted to represent the fact that it now had 3 locations, off, fw and bw. Secondly, the inputs were doubled for both the downstream and upstream conveyor. Each conveyor exports some variables and events for use specifically for the downstream and some for the upstream conveyor. In the case of the bi-directional conveyor, both bordering conveyors export both sets of variables and events. Using the input prescribing the drive direction, algebraic variables are used to couple the right set of inputs to the downstream and upstream conveyor. For instance, for the Transfer signal this is: \[
\text{alg bool prev\_Transfer = if i\_dir = fw: prev\_Transfer\_fw else prev\_Transfer\_bw end;}
\]
These changes to the model make sure that the motor signal functions correctly and that the conveyor responds correctly to downstream and upstream signals. Figure 4.11 shows an overview of input interface.

![Figure 4.11: Overview I/O interface bi-directional TRS definition with up- and down-stream conveyors](image)

Influence on Tracking

Tracking also needs to be adjusted. There are two major options. One options is that the speed of the conveyor becomes negative and products start at the end of the belt and transfer downstream when they get down to zero. The other option is flipping the axis on the conveyor and keeping the zero coordinate always bordering the upstream conveyor. When the direction is switched, a function is executed reorienting the tracking array. The first option is more intuitive and does not require any specific transition. However, most of the variables interacting with tracking use the fact that products move from 0 to the end of the conveyor. Simply starting to move products backwards, requires expanding
4.3. Reuse

these variables to account for the direction of the conveyor. For these reasons, the second concept of always keeping the zero coordinate of tracking facing the upstream conveyor in tracking is implemented. Figure 4.11 shows the interface of a bi-directional conveyor with its neighbours and illustrates the principle of flipping the axis in tracking. The fw and bw indicate when the input or setting shown is relevant.

Listing 4.7 shows the function used to adjust the tracking coordinates to the new axis orientation.

```plaintext
type prd = tuple(real hd, tl; int br_id);
// Function capable of reordering the list in case of belt reversal on Bi-Directional conveyors
    int i = 0;
    prd save;
    // Update tracking array with value of dx
    tracker := updatelist(tracker, n, dx);
    // Starting at product 0 and product n-1
    // the function switches the product info
    // While moving, the coordinates are adjusted
    // to the new axis
    while i <= (n-1)/2:
        save := tracker[i];
        // The head becomes the tail and vice versa
        tracker[i] := (1 - tracker[n-1-i][tl],
                       l - tracker[n-1-i][hd],
                       tracker[n-1-i][br_id]);
        tracker[n-1-i] := (1 - save[tl],
                           l - save[hd],
                           save[br_id]);
        i := i + 1;
    end
    return tracker;
end
```

Listing 4.7: Function used to change the orientation of tracking

Separate model definition

Due to the excessive amount of extra inputs which would go mostly unused in the standard TRS, this model definition was separated from the standard definition. Re-using a definition for a different function variant should not inconvenience using the definition for its main purpose. This model definition has been tested and validated in CIF, but has not yet been validated in a real-time implementation.
4.3.2 Master and Slave Configurations

In certain situations, it is required that a group of conveyors move in unison, without stopping. This can be the case if measurements are being conducted, or if products can reserve free conveyor space in advance, as in Subsection 4.2.2. This is referred to as a master-slave configuration. The most downstream conveyor is the master conveyor, all other conveyors imitate the state of this Master conveyor and are referred to as Slaves. In regard to conveyors downstream and upstream of the group under master-slave control, the conveyors function as one conveyor. It should remain possible for these conveyors to function independently.

It should be possible to couple a group of conveyors and dynamically appoint a master. To adjust the TRS model definition for such outside control, extra inputs containing the Master state are required. Furthermore, certain signals should be bundled and taken into consideration by the master. For instance, the master should not take all the conveyors into an energy conserving mode while other conveyors have products in tracking. When a single conveyor is in error or turned off, all other conveyors should copy this behaviour.

Implementation in the TRS Definition

Preferably, this concept is implemented without disturbing the capacity of every conveyor to work independently. To the TRS, an extra input is added which represents the Master state and an input boolean which indicates the presence of a Master conveyor. A switch is built around the internal status automaton by altering a few strategic boolean variables. If the master signal is true, these variables rely on the Master state, otherwise on the internal state variable. An example of this switching behaviour is shown in Listing 4.8.

```alg
// Master–Slave Variables
// The State of the conveyor is switches on the input boolean master
// The value of the state is either the input state masterstate
// or the state computed in the status automaton
alg TRS_state ms_state = if not master: status.state
else masterstate
end;

// This ms_state is then used to calculate for instance the motor output
alg bool MTR = (ms_state = running or ms_state = starting_up);
```

Listing 4.8: Master slave switching behaviour

With these two extra inputs, each conveyor can function as a slave conveyor while the master signal is true. If the master signal turns false, it can immediately continue as an individual conveyor.
**Master-Slave Controller**

In order to provide the new input signals, a new model definition is introduced. Incorporating a scalable solution to compute master signals in the TRS model is possible, but in most conveyors this functionality would never be used. A separate small model definition for a master controller is created. This master controller must take into account the state of every conveyor. If one conveyor is in error or stopped, the Master state should represent this regardless of the state of the Master conveyor. Furthermore a signal must represent if every conveyor is ready to go to an energy conserving mode. As the number of conveyors varies, simply giving every state as an input is not a scalable solution. However, by taking a boolean from each conveyor indicating if it is in error and combining these in an OR construction like \( \text{TRS}_x\text{\_state} = \text{error or } \text{TRS}_y\text{\_state} = \text{error or } \ldots \), one signal can represent a scalable amount of conveyors. The resulting automaton definition is shown in Listing 4.9.

```plaintext
// Variables used as input for the TRS
alg bool master = i_masl; // Boolean depicting whether master/slave is on or not

alg TRS_state masterstate = if in_error: trs_error // There is a belt in error
elif in_stop: stopped // There is a stopped belt
else TRS_masterstate
end;

alg bool CE_ok_ms = CE_ok_ms_in;
end

// Instantiation
ctrl_masl : masl_ctrl(TRS_1.in_error or TRS_2.in_error,
TRS_1.in_stop or TRS_2.in_stop,
TRS_1.CE_ok and TRS_2.CE_ok,
master_slave_active,
TRS_2.status.state);
```

Listing 4.9: The Master-Slave Controller

In conclusion, the status automaton of the Master conveyor continues to function and exports its status to a Master controller. This controller combines this status with information regarding the slave conveyors and exports a Master state and a Master boolean.
When the Master boolean is true, the Master conveyor can only enter an energy conserving mode, if the combined signal is true.

**Incorporation in the TRS Definition**

This solution only adds three inputs to the model definition of the TRS and within the definition, the number of changes are minimal. In the case that a conveyor is not in a Master and Slave configuration, these three inputs can be given dummy parameters blocking the functionality. As it has a relatively small influence on the standard conveyor and is expected to be combinable with most other function variants, it is included in the standard TRS model definition.
Chapter 5

Real Time Implementation

The goal of this project was to create a model definition library capable of controlling the BHS of a small airport. In this chapter, a BHS controller is built using the new and adjusted model definitions. While new functionality can be tested through interactive simulation in CIF, this does not automatically mean that it works correctly in real-time on PLC. Errors can be introduced by the transformation and differences between execution on PLC and CIF. Thus, in addition to the interactive simulation, a test is carried out on a PLC using emulation. Emulation of a BHS is achieved using a computer emulating an airport, interfacing with the I/O of the PLC. In the last section, a test plan is formulated and software requirements are formally tested.

5.1 Controlling a Small Airport

Before instantiating the controller, the first step is to model in- and output zones. The second step is to determine the criteria for the test setup and to choose an available emulation model. Once the specifics for the BHS are known, the control and hardware models can be built using the definition library. Finally, using interactive simulation, the basic functionality of the controller can be tested in the CIF environment.

5.1.1 Modeling the In- and Output Zones

The in- and output of the system, the Check-In Zone (CIZ) and the Carrousel (MTZ), have not yet been discussed. In this section the important design choices and differences of these zones are explained.
Check-In Desk

The Check-In Desk is the input of the system, and the first zone encountered which is actively operated by a person. Check-In Desks come in several variants, consisting of 1 to 3 conveyors. On the first conveyor, the product is weighed, labeled and then manually dispatched. It is the responsibility of the operator that these steps are carried out correctly. After manual dispatch, the zone and section controllers take over, moving the product down any following conveyors, running any additional tests and interacting with the Collector Zone to deposit the product. In the first step of labeling and weighing, the conveyor can be moved by pushing a button. When the product reaches the PEC on the first conveyor, it should stop and wait for dispatch. From this behaviour, 5 basic modes with distinctly different behaviour can be determined which are applicable to all conveyors in Check-In Desks.

- **off** Zone is not in use
- **idle** No products in tracking before the PEC and no outside input to move
- **moving** Products in tracking before the PEC or outside input to move
- **r4dispatch** Product in the PEC awaiting the chance and/or order to dispatch
- **dispatched** Dispatching a product to the next conveyor, either inline or as an infeed

As these modes are very different from the modes of the standard conveyor, and other functionality in the standard TRS is not required here, a separate check-in desk conveyor section was created. The tracking and PEC handling events are identical to the TRS. However, the status automaton reflects these 5 modes in 5 locations and moves between these locations waiting for hardware or user inputs.

Because of the different variants and extendable functionality, a set of boolean inputs is given to each conveyor detailing the functionality required. For instance, the first conveyor on which the product is initially placed should wait for user input and the availability of the subsequent conveyor. The second or third conveyor only has to wait for the signal of the downstream conveyor. By a boolean input variable, this difference is reflected in the code. A simplified example is shown in Listing 5.1 and Figure 5.1, where instead of a set of booleans, only one boolean is given as input indicating whether dispatching is automatic or manual.
5.1. Controlling a Small Airport

Listing 5.1: Example of controlling functionality through input booleans

By combining a number of booleans such as bool automatic_dispatch and providing them in a tuple as input, each CI can be customized to a specific conveyor in a specific zone variant.
A number of additional smaller automata take care of individual requirements. For instance, when a product is permitted to dispatch and the zone is turned off, it should continue to try and deposit the product. Only when it does not succeed within a certain time frame, it is allowed to turn off. These requirements are ideal to be modeled in small automata, blocking and allowing the relevant events. An example is included in Listing 5.2 and Figure 5.2.

```plaintext
automaton dispatch_timeout:
    // Automaton handles time out during dispatching
    // when the CIZ is turned off
    // If a product is waiting for dispatch
    // it will continue to try for one minute
    cont t = 0 der 1;
    disc real disp_timeout = 60;
    // disp_pending is an algebraic variable on a higher scope
    // indicating whether a dispatch
    // is still pending in this conveyor or a preceding one
    disc bool disp_pending;
    // Input Boolean i_on is the zone on/off signal
    disc bool i_on;
    // Event e_off is triggered elsewhere when i_on turns false
    event e_off;

location idle:
    initial;
    // blocking e_off
    edge e_off when not disp_pending;
    edge when not i_on and disp_pending do t := 0 goto counting;

location counting:
    // Allow e_off when time has run out
    edge e_off when t >= disp_timeout goto idle;
    // Product has dispatched before time ran out
    edge when not disp_pending or i_on goto idle;
end
```

Listing 5.2: Automaton handling a separate requirement
Finally the Check-In Zone must be compatible with the windowing algorithm discussed and implemented in Subsection 4.2.2. The requirements for the infeed are discussed in that subsection and an automaton is created adhering to these requirements.

Carrousel

The Carrousel is a long oval conveyor which forms a complete circle. Each destination in the BHS ends in a Carrousel from where the luggage can be moved to the correct airplane. Functionality-wise, this zone is not very complicated. It consists of a single TRS leading to the Carrousel itself. The Carrousel is a simplified version of a TRS as it has no general tracking or downstream conveyors. A number of infeeds form the upstream conveyors.

The model of the Carrousel is for this reason a very simplified version of the TRS. Its status can only be on, off or error. It receives product tuples, but only uses the data to delete the BR record for the product. For the infeed to the Carrousel, some modeling is required. This infeed has be to be controlled in order to avoid collisions. A PEC is placed on the carrousel, upstream of the infeed. From this point onward, local tracking is implemented, using the information from the PEC. From the local tracking, an able2receive variable as seen in Figure 4.5 is created for the infeed. Using this construction, the infeed can be a normal unadjusted conveyor.

5.1.2 Airport

In this section the required functionality of the test setup is specified. In order to test data communication, there should be a point where data is written to a BR Record and a point where a sorting decision can be made based on this data. Furthermore, it should
demonstrate that an entire airport can be controlled, providing the path from check-in to
carrousel. The following requirements are defined:

- The BHS for the airport should run on a single PLC
- The PLC should control a departure line from check-in to carrousel
- The PLC should control at least one measuring object
- The PLC should have some sort of sorting option
- The PLC should control at most 60 sections because of cycle time and memory
  restrictions

Using these requirements, a suitable Test Line was chosen from the available emulation
models. An overview is shown in Figure 5.3. All the non-standard zones are highlighted.
It can be assumed that any conveyor not highlighted is part of a transport zone. Using
three check-in desks, products are fed into the system. From the collector zone, products
move through a transport zone to an identification zone (IDZ) which has a BSO. After
the IDZ, products are screened using a hold baggage screening object (HBO) in a hold
baggage screening zone (HBZ). From the HBZ, products are transported to a vertibelt
zone (VBZ), which sorts the products according to the security results of the HBO. Secure
products are allowed to pass to the carrousel, all other products are diverted to the side
conveyor and leave the system.
This configuration makes it possible to test all the newly designed and redesigned
zones. Furthermore, the communication protocol as designed in Chapter 3 is fully tested.
Products enter data tracking once they leave the Collector Zone, from here on they will
have a BR ID and an accompanying BR in the array. Both the BSO and the HBO interfaces
will update the BR. Lastly the BR is accessed by the VBZ for the sorting policy. Using
this Test Line, the new and old functionality is all represented and can be shown to be
correctly modeled.

5.1.3 CIF Control Software

Using the information available on the Test Line, a set of parameters and coordinates
can be gathered of each conveyor. Using the specifications, the controller and hardware
representation of each conveyor can be initiated in respectively the control and hardware
files. As the sections are the only system level requiring physical parameters such as
length and PEC locations, the rest of the system can be initiated. This yields a controller
and a hardware model, which can be merged to form an interactive visualization. Using
this visualization, the model can be tested and validated in CIF.
During development, all of the model definitions were already tested separately and
combined in smaller loops, as such no major bugs were expected during testing in CIF.
However, it was not immediately possible to test the model extensively, because of
5.1. Controlling a Small Airport

Figure 5.3: Overview of the Emulated Test Line
difficulties with the visualization of the CIF simulator. The control model developed in this report was the largest CIF specification to date. The interactive visualization was not efficient enough to deal with such large canvasses and images, while simulating in real-time. Furthermore, as the interactive visualization is not equipped for zooming or scrolling, a choice between individual conveyor details or an instantaneous global overview had to be made. The choice was made to scale the system so, that such details are still visible and can be clicked. However, in this way the whole conveyor system cannot be visualized at once. The system was divided over three separate visualizations running in parallel, each subvisualization can be navigated using the side-bars. As each sub-visualization can be tightly fit around that part of the system, the sum of the three sub images is smaller than the original image. Furthermore, efficiency enhancements in the language and the CIF simulator allowed the model to run at 4 fps. Due to the large amount of memory involved, a computer with a 64-bit operating system and at least 2.3 GB of memory is required to run the visualization.

With these improvements, the model could be simulated sufficiently to test basic flow and ensure that instantiation of definitions was carried out correctly. Basic conveyor interaction as well as the required BR functionality and object interfacing could also be tested using hardware representations of these objects. However, as both the control and hardware representations use the same variables as input for system parameters, errors in relation to the specifications for the emulation model are not caught. As the visualization is by far the most expensive part of the simulation, automatic tests could be run at much higher speeds without the visualization. While the visualization is a very powerful tool for simulating subparts of the system and testing new model definitions very early in development, it is not very usable for such large systems. A more efficient visualization with scalable images would be much easier to use. Furthermore, dynamically gaining access to the right variables during a simulation is very important for testing, but access can only statically be specified before the visualization is started.

5.2 Transformation and Implementation

This section elaborates on the transformation from CIF to SCL and the implementation of the transformed code in Siemens TIA Portal. Siemens TIA Portal is a PLC software development environment capable of connecting to a PLC. It can upload and download software to the device and monitor the PLC while it is operating.

5.2.1 Transformation

In the transformation, a specification based on hybrid automata and synchronous transitional behaviour is transformed to a sequential program of which the behaviour is at least a subset of the original program. The removal and modification of these language properties have a sound theoretical basis, which is not elaborated on here. However, a
short, step-by-step overview is given. The transformation from CIF code to a Siemens SCL program is actually a series of model conversions, an overview is shown in Figure 5.4. The first step normalizes the model definitions by encoding automaton locations as variables, creating a different but equivalent CIF specification. This way, each automaton becomes a single-location automaton with a number of self-loops. Each self-loop altering the location variable is used to go to a different location in the original automaton. The concept of a normalized automaton without synchronizing events can be implemented in all imperative programming languages while locations are mostly exclusive to automata-based languages.

The second step eliminates synchronous behaviour and results in a specification which is no longer valid in CIF. The problem with synchronous behaviour is that a synchronizing transition changes variables and locations in multiple automata as well as depending on variables in multiple automata. Furthermore, CIF only takes into account the old state when computing the new state. No intermediary states are calculated. Sequential languages execute programs line by line and after every line the program is in a new intermediary state. Subsequent lines or actions deal with this state. This transformation step must turn the CIF program into a list of sequential actions being a subset of the original behaviour. A simplified explanation of this step is that the CIF+ specification is altered so that a synchronizing transition is removed from all but one of the synchronizing parties. In that automaton, the event can only take place if the conditions of all the original synchronizing parties are met. Changes to the variables of the other synchronizing parties are also initiated in this automaton. As this is a violation of the Local Write/Global Read property, the CIF+ specification is no longer a valid CIF specification. From this CIF+ specification the code can be transformed to any sequential programming language, in this case a CIF+ to SCL transformation is implemented. For a more theoretical explanation of the second step, look to [31].

5.2.2 Siemens TIA Portal

While the transformation to PLC code delivers working SCL code, this it not the last step in the implementation. The software needs to be linked to the right in- and outputs and needs to be downloaded to the PLC. This is accomplished using a PLC software development environment, TIA Portal [30]. TIA Portal is a Siemens product, intended to integrate earlier and separate products into one framework. In this project, TIA Portal
version 13 is used. Figure [5.5] shows an overview of TIA. TIA Portal is intended as an environment for the development of PLC programs. However, in this project the generated code is imported into the program automatically, generating the specified blocks. This is the first indication about possible errors or discrepancies in the generated program, as reported by the SCL compiler. The generated blocks are not finished yet, some adjustments must be made by hand. As the software is developed in CIF, the controller and the hardware are not properly linked. The CIF controllers expect inputs, and variables are declared for these inputs, but an additional file is needed to link these input variables to the incoming addresses. From the documentation on the Test Line, the I/O links can be acquired. A function block is created, linking each input variable to the value of the corresponding input address. A similar function block couples variables in controllers to specific output addresses. An example linking a boolean PEC input in a controller to the input address in the PLC is "TRS_1_PEC": "I_0003_28_01_71_PEC_EOS". Every time the cyclic program runs, the very first thing it should do is run the inputs function block refreshing the input variables. Similarly, the last thing it should do is executing the outputs function block, writing the new values to the output addresses. The function blocks linking the I/O need only be specified once, calls to these blocks in the main OB need to be added again for each new version.
5.3. Emulation Testing

Once the program has been adjusted and can be correctly compiled, it can be downloaded to the PLC. TIA Portal can connect to a PLC and download or upload software. Once the software is loaded and the PLC is running, TIA Portal can be used to check the PLC diagnostics, look at the cycle time, and the memory usage. In short, it can be an interface between the user and the PLC. Using TIA Portal, variables in data blocks can be monitored and values of I/O addresses can even be forced to take a certain value. Using this functionality, TIA Portal is used to coordinate tests and validate the correct working of the program.

5.3 Emulation Testing

This section deals with the initial implementation of the software on the PLC and the first phase of testing, debugging. As both the software and the transformation contain new concepts and data types previously untested, the source of errors is not always immediately clear. An explanation on the concept of testing using emulation is given, as well as an overview of the issues encountered in the transformation. Finally, an explanation of the procedure of connecting to objects on the PLC as applied in this project is given.

5.3.1 Emulation and PLC

While in the previous projects the control programs were tested using actual hardware, this project requires too much functionality to be incorporated in a test setup. Instead of a hardware loop, the PLC controls an emulation model on a computer. The computer in question runs a Windows OS and has a PROFIBUS connection. PROFIBUS is a communication network for industrial processes, comparable to the ethernet protocol [13]. In this case, PROFIBUS is used to establish an I/O network with the PLC over which all the communication between the emulation model and the PLC is directed. Using a 3-D simulation tool such as Demo3D or Automod, a BHS can be modeled and simulated [7][11]. By connecting the inputs of the PLC with simulation output and controlling the simulation with PLC output, the Low Level Controls (LLC) can successfully be emulated. A screenshot of the emulation model, featuring the IDZ, HBZ and the sorting VBZ, is shown in Figure 5.6
Using emulation, large and complex systems can be tested, while no actual equipment is required. Naturally, actual hardware introduces more uncertainties and has more possible factors for error. Regardless, with emulation a large number of scenarios and basic flow functionality of a new BHS can be tested, without the complications of having to build and maintain an expensive setup or having to go to site.

### 5.3.2 Debugging the Models and the Transformation

During the debugging phase, the errors encountered could be divided into three categories.

- Instantiation errors caused by the manual input of the system constants and the manual linking of in- and output in the PLC.
- Behaviour different from the CIF specification introduced by the transformation process
- Modeling errors in CIF

The first category of errors can be found if individual instantiations are malfunctioning, instead of globally. A conveyor reporting many missing products might indicate an error in the PEC location or the mapping of the PEC input. Similarly, a conveyor might not start because the motor output is not mapped to the correct variable and so on. These errors are easily fixed and quickly identified, but the high volume makes elimination very time consuming.
As the transformation is still in development, not everything is transformed correctly. Currently, the largest issue is sequencing variable updates in an incorrect order. In CIF, it does not matter in which order variables are updated during a transition. The values of variables are always taken from the previous state of the system, so before the start of the current transition. In PLC, the order is important and a different order might lead to a different outcome. An example of two types of updates which lead to different behaviour is shown in Listing 5.3. Listing 5.4 shows the current outcome of the transformation of this CIF specification, Listing 5.5 shows the correct outcome.

Listing 5.3: Original CIF specification

Listing 5.4: Incorrectly transformed PLC specification

Listing 5.5: Correctly transformed PLC specification
Chapter 5. Real Time Implementation

// Correct transformation outcome
// Event e_event_1
IF TRUE THEN
  // Temporary variable required
temp := x;
x := y;
y := temp;
END_IF

// Event e_event_2
IF TRUE THEN
  // Shuffling the order is sufficient
y := x;
x := 2;
END_IF

Listing 5.5: Correctly transformed PLC specification

Finding the transitions leading to incorrect behaviour is very time consuming. If it is caused by a dependency of 2 variables in one edge as shown above, it can be remedied in CIF by changing the order of updating in the edge. However, if it is caused by an dependency between two synchronizing edges, it cannot be fixed in CIF and has to be altered in a post-transformation step. Having the transformation compute the correct behaviour is not trivial and was not achieved during the testing phase. However, in a future version a solution will be implemented. The current version already indicates edges which could show incorrect behaviour.

The third source of errors were modeling errors in CIF. These were relatively few, as most were already caught using interactive visualization. The errors occurred in untested scenarios. For instance, during testing in the CIF environment conveyors always became inactive if no products were on the belt. Initially during emulation, this functionality was disabled, this caused the occurrence of an error in the Collector Zone. Such unanticipated scenarios will continue to occur as testing everything manually is prone to human error. The implementation of some form of automated testing sequence or the enforcement of a testing plan would cause such errors to be caught.

Of the error groups described, errors originating in the CIF models and some of the code leading to incorrectly transformed behaviour can be permanently solved in the CIF models. However, incorrect updating sequences because of updating dependencies between two automata have to be fixed for every new version. The following post-transformation steps had to be taken.
5.3. Emulation Testing

1. Set the length of arrays in User Defined Types, as these are often incorrectly given a higher maximum index, leading to memory issues.

2. Change the variable type string to char, as characters are not native in CIF but strings use too much memory in PLC.

3. Replace transformed transitions with incorrect sequences for updating variables with correct versions.

4. Change initialization of the BR Array, which is executed inefficiently, using more than the available temporary memory.

Finally, there are some changes which have to be made to the model, such as calling and executing the I/O mapping. These cannot easily be eliminated as hard-coding these changes would force the user to always to use identical file names and procedures for the I/O mapping. Such changes will have to be made until an alternative or automatic I/O mapping is implemented, but making these changes takes very little time.

At the time of writing, Point 1 and 4 have already been eliminated. A solution for Point 3 is in development. By calculating dependencies between updated variables in transitions, a correct updating sequence could be computed. Point 2 is difficult since it can depend on the specific implementation. However, simple measures can be taken to restrict the amount of memory a string uses in the PLC.

5.3.3 Connecting to Objects

In section 3.2 an interface for communicating with objects was created. In this section, the actual linking of this interface to the objects and reading incoming messages is discussed. In CIF, this was modeled as an input, a single variable of type string. In a PLC, this message is encoded by a range of input bytes, each byte being one character. The whole range of bytes forms the incoming message. In order to read this message, an array can be made, holding these characters in the right sequence. In the I/O mapping, each index of this array must be linked to the right incoming byte. Similarly, for outgoing message such as product information going to the screening machine, a range of outgoing bytes must be written from a single message variable in the interfacing Function Block.

Baggage Scanning Object

The BSO sends one message, but receives no messages. The message it sends are 30 characters and contain the three label codes with the highest read count. A new message is announced by flashing an input bit. When this bit is flashed, the interface loads the new message. The transformation alters input variables to functions. These functions call the required values from the I/O mapping using tags. A tag is a pointer to a specific bit, byte or variable. As an array is not a single variable, a tag cannot point to it and the
standard method does not work. By adjusting and replacing the function, the correct message is copied. Currently, this step has to be performed for every new version. A small example in Listing 5.6 shows the concept.

```
FUNCTION "BSO_mess" : UDT3
BEGIN
  // Initial function
  // Tag "BSO_Message" cannot point to an entire array
  #BSO_mess := "BSO_Message";
  RETURN;

  // Replaced function
  #BSO_mess[0] := "BSO_Message_Char1";
  #BSO_mess[1] := "BSO_Message_Char2";
  #BSO_mess[2] := "BSO_Message_Char3";
  #BSO_mess[3] := "BSO_Message_Char4";
  RETURN;
END_FUNCTION // BSO_mess
```

Listing 5.6: Original and replaced function retrieving the BSO message

There are more efficient methods of mapping a range of inputs on an array. For instance, automatic mapping by giving a start address and having the array overlay all the following addresses. For now, the easiest implementable solution was chosen. As the concept has been proven to work with transformed CIF code, more efficient methods could be tested and implemented in the transformation. If the transformation could detect if an input variable is an array, the automatic mapping could be implemented. The automatic mapping requires only the initial address to be set, no additional actions would be required from the user.

### Hold Baggage Screening Object

The HBO sends a message in a similar fashion as the BSO, but also receives a message from the BHS. As the message sent by the HBO is handled in the same way as for the BSO it is not discussed here. The message received by the HBO consists of some bag identifiers. Once the HBO requires this data, it flashes a “request message” bit. If this bit is on, the interface takes the data for that product from the BR Array and composes a message. When the message is finished and the output addresses are overwritten with the new message, a “message sent” bit is set to true. In CIF these variables were merely written and not used. In the PLC program, they have to be relayed to the HBO. In the output mapping, the new outgoing message bit is linked to the right variable in the interface. The individual bytes of the output range reserved for sending the message are overwritten by values of the outgoing message, basically the opposite action of the replaced function in the example above. As this is part of the manual I/O mapping, it only has to be performed once.
5.4 Testing and Performance

In the previous section, the debugging phase of the real-time implementation was described. In this section, the testing phase is elaborated on. The difference between debugging and testing is that during debugging, the user is aware that there are still bugs present in the software. In the testing phase, the user assumes the software to be bug-free and devises a test plan to prove this assumption. If during these tests an error is discovered, it is fixed and the tests are restarted from the beginning. When the test plan can be run without any unexpected or incorrect behaviour, the testing phase is complete.

5.4.1 Test Plan

In creating control systems using models, two main sources of errors are expected to be found. Errors because of incorrect modeling of requirements in the model definitions, and errors because of instantiation of the model definition with incorrect parameters. Incorrect instantiation can have many sources.

- Incorrect information on system parameters.
- Incorrect linking of definitions.
- Incorrect coupling of I/O addresses.

However, most of these errors are either instantly obvious during debugging or mostly inconsequential. For example, if an error is made with a PEC location, placing it one meter off, tracking will not work at all. If on the other hand, if the error is only one centimeter, it falls inside the error margin and has no consequence. Furthermore, these instantiation errors can be eliminated with further automation of the design process. For this reason only modeling errors, are taken into account.

Even when only considering modeling errors, the number of requirements to test is very high, dozens of requirements are implemented for each controller. Furthermore, there are different levels to each requirement. An official requirement might state that a certain mode should be triggered if the downstream conveyor is unavailable. In the implementation, this requirement might be split into 2 or 3 separate requirements. Enter that mode if ..., leave that mode if ..., and so on. Ergo, there is a difference between the written requirements in the product book and their actual implementation. Moreover, a requirement states that something should always happen. A test scenario for a requirement only shows that it is validated for that specific scenario, it does not prove that there is no series of event or circumstances in which that requirement is not fulfilled.

CIF offers the possibility to formally specify requirements in the form of automata. There are two major options for interpreting requirements in the form of automata. It is possible to indicate how the system should function. These automata can be directly implemented.
Chapter 5. Real Time Implementation

as (part of) an automaton in the actual CIF control definition. Such requirements are a
great starting point for modeling new control definitions or can be taken almost directly
from already existing CIF specifications. However, during testing it is less important to
know how a requirement is implemented. The focus is on finding out if requirements fail. If the tester can see from the requirement which variables to check and in which
locations certain behaviour indicates a failed requirement, this is more meaningful than
a requirement explaining how it is implemented. This kind of automaton might also
be used in run-time verification for automatic detection of failed requirements. As it
concerns the testing phase, the approach used in this chapter detects when a requirement
fails. In order to illustrate the two approaches, two examples are shown for the same
requirement: When starting up, a conveyor should run its own length once. During this
time, the PEC will detect any products on the belt.. Both methods are shown in respectively
Listing 5.7 and Listing 5.8. Figure 5.7 and Figure 5.8 show the graphical representations.

Listing 5.7: Requirement starting_up as modeled

```python
// Alg bool v represents the speed of the conveyor
// If the conveyor is stopped v = 0 else v = speed
automaton def Requirement_start_up_1(alg real v, conveyor_length;
    event e_start, e_stop):
    // Displacement of the belt
    cont dx = -1 der if dx > -0.5: v else 0 end;
    monitor e_start, e_stop;

    location idle:
        initial;
        edge e_start do dx := 0 goto starting_up;

    location starting_up:
        edge when dx > length do dx := -1 goto idle;
        edge e_stop do dx := -1 goto idle;
end
```

Listing 5.8: Requirement starting_up as modeled
5.4. Testing and Performance

Figure 5.7: Automaton overview of Listing 5.7

---

```plaintext
import "states.cif";
// Alg bool v represents the speed of the conveyor
// If the conveyor is stopped v = 0 else v = speed
automaton def Requirement_start_up_2(alg TRS_state state;
alg real v, conveyor_length;
event e_start, e_stop):

    // Displacement of the conveyor
    cont dx = -1 der if dx > -0.5: v else 0 end;
    monitor e_start, e_stop;
    location idle:
        initial;
        edge e_start do dx := 0;
        edge e_stop do dx := -1;
        edge when dx < conveyor_length and state = running
go to requirement_failed;
        edge when dx > conveyor_length and state = starting_up
go to requirement_failed;
    location requirement_failed;
end
```

Listing 5.8: Automatic detection of Requirement starting_up failure
In order to establish a basis for following projects, a list of about 40 requirements spread over basic flow functionality, new zone functionality and object/BR functionality was compiled and a series of experiments was setup which tests all of these requirements at least once. This test plan is included in Appendix A.

5.4.2 Test Report

Using the plan devised in the previous section, the Test Line setup was tested. The testing method was carried out by sequentially running the defined experiments with the following error handling method.

- Run the experiment
- If a requirement is invalidated:
  - Abort experiment
  - Fix the failing requirement in the model
  - Restart from the first experiment
- Else:
  - Run next experiment

As each change to the code might lead to new errors, after every bug fix the earlier requirements should also be revalidated. In the same manner, after adding new features or making small changes, the control software needs to be retested. Testing continues until all experiments can be run in sequence without the occurrence of errors.
The test plan is executed twice, once in CIF and a second time on the PLC. If the software in CIF is validated, any errors in the PLC software must originate in the transformation. If this is found to be the case, then a property-preserving transformation would transform validated CIF software to validated PLC software. Of course there might be errors which can never be found in a simulated environment. However, validated transformed software would require a shorter real-time testing phase as less iterations are required.

Although the system appeared to be bug-free in the previous testing phase, some errors were encountered during the testing runs. In CIF, some errors were encountered due to changes to the Check-In Desk, after which the model definition was only partly retested. In PLC, variables in some events were still updated in incorrect sequences, causing small requirements to malfunction. In both CIF and PLC, after adjusting the requirements and a couple of test runs, the test plan could be carried out without issue.

In this project, a clear distinction was made between a debugging phase and an official testing phase. If in the future the transformation is complete and a validated and correct PLC program is generated from a completely tested CIF specification, the argument might be made that a debugging phase is no longer required. However, a difference between the ideal environment of the simulated hardware and the actual hardware, even using emulation, will always remain. In this project problems were encountered due to fire doors which were not modeled and ignored in CIF but were invisibly modeled and automatically closed in the emulation model. These fire doors had to be accounted for by permanently opening them. Such unanticipated differences between simulation and actual implementation will, in general, be present. A testing phase only focusses on system requirements. A short debugging phase in which the tester actively looks for system malfunctioning instead of assuming correct behaviour will always be beneficial.

5.4.3 Analysis of System Performance and the PLC Code

As explained in Subsection [1.1.4], Vanderlande requires PLCs to be able to control 200 sections, while retaining a certain maximum cycle time. Naturally, one of the end goals in using CIF combined with code generation is achieving the same cycle time. In previous projects, using previous versions of the code generation process, this goal was not met. While the current generation of the code generation is much more efficient in some ways, due to the recent implementation of channel functionality and the extension of the tuple functionality, a lot of unnecessary temporary variables, variable assignments and function calls are used. The implementation of more efficient algorithms in future versions will lower the amount of temporary variables, variable assignments and function calls. Moreover, the functionality of the CIF model itself was extended, increasing the data flow and the total amount of variables due to the implementation of channel and BR functionality.

The cycle time of the PLC was measured between 20 and 30 ms with about 40 sections, which does not meet the Vanderlande cycle time requirements. However, despite of
the added functionality, the result is similar to the previous project, where a cycle time between 20 and 30 ms was achieved with about 40 sections [17]. As the functionality of the system is extended, a comparable cycle time means an efficiency improvement for the transformed code.

The cycle time seemed to be dependent on the product load, more activity generally led to higher cycle times. While this is to be expected, it may partly be caused by inefficient use of variable assignments and function calls. While the initial variables required to assess the condition for a transition must be checked every cycle, the generally much higher amount of assignments and calls occur if this initial condition is satisfied. A rise in product load increases the amount of eligible events and might cause a rise in cycle time.

The modularity and readability of the code has been improved in some ways and has been decreased in others. On the one hand, the number of functions is smaller, instantiation is easier and requires less input. A global data block is established centrally holding parameter values. The readability of the code has drastically improved as events are now only shown between two of the specific possible locations instead of combining all in one piece of code. On the other hand, the implementation of channels created some complications regarding modularity and readability. In CIF, channels are very dynamically usable even when defined in model definitions. As such, a generalization is difficult to achieve in the transformation. For this reason, channels operating between model definitions in CIF are handled outside of the modular function blocks in the PLC. Separate FBs are introduced, all handling nothing but such channels. Switching out a function block for a model definition with a new version only works if the interacting channels are not adjusted in any way. Otherwise, the code handling the execution of the influenced channels must be changed as well. This negatively affects both the modularity and readability.

Currently, there are still plenty of opportunities for efficiency improvements in the transformation. Elimination of unnecessary temporary variables, variable assignments and function calls is in development. Replacing as many real variables as possible with integers would mean a large improvement, saving up to a factor 4 on computation time [8]. By continuing to improve the transformation, the cycle time of the transformed controllers can continue to improve until the Vanderlande requirement might be met.
The goal of the project was to develop a control library in CIF, capable of controlling the BHS for a small airport using a PLC. This would integrate in- and output zones, basic sorting and flow, and data tracking into the existing control models. The following subgoals were identified.

- Implement data tracking.
- Improve the existing models.
- Design the required zones (Input zones, Output zones, and Measuring Zones).

Data tracking was implemented in the form of a globally accessible database. This database consists of an array in an automaton, and holds a set of information for each product in data tracking. The Global Read/Local Write property of CIF does not allow for direct alteration of this database by other automata. Instead, these automata indirectly send new information via channels. As all updates originate from measurements carried out by measurement equipment, the number of update types is related to the different types of measurement equipment. Each type of measurement equipment is assigned an update type and uses its own unique update channel.

Existing models were evaluated for the scalability, possibilities for interfacing, and the capability for reuse. With scalability, the goal was to provide more scalable solutions, making model definitions more versatile and applicable in more configurations. Interfacing focussed on the interaction between definitions, specifically between conveyors. The conveyor model is the most extensively used definition and exists in many possible situations. A generic and versatile interface is very important. Reuse of definitions looked at new function variants of model definitions. For instance, the bi-directional function
variant of the standard TRS. New features can be implemented as standard functionality, or in the form of a new and separate model definition. The decision to incorporate the new functionality in the original definition or create a new one, was made based on the impact of implementation on the original definition.

All the required zones were designed according to the requirements in the product book. Input zones consisted of the Check-In Desk and the Collector Zone. These zones were created, and they interact using a simple windowing algorithm. The output zone is the carrousel, a long oval conveyor which forms a destination for the BHS. From the carrousel, products are moved to the correct airplane. The carrousel conveyor is a simplified version of the actual conveyor, as most functionality is not required. The measuring zones were the Identification Zone and the Baggage Screening Zone. The Identification Zone scans product labels, which are used to identify the product. The Baggage Screening Zone uses X-ray to screen products for security threats and forbidden content. Both objects are communicated with via an interfacing process. These interfacing processes send new information to the BR Array using the specific update channel defined for that update type.

In order to test if the control library is capable of controlling the BHS of a small airport, a test was conducted using an emulation station and a PLC. An existing emulation model was chosen. According to the specifications, a CIF controller was assembled from the definition library. In the CIF environment, the interactive visualization was used to test the created CIF controller model. Using the code transformation, the CIF specification was transformed to a PLC program. After a debugging phase of the transformation and the resulting code, a testing phase was conducted according to a test plan. A set of requirements was compiled, and a series of experiments was set up for both CIF and PLC. After the CIF test round validated all the requirements for the CIF controller, it was transformed to a PLC program. The PLC test round found no errors which were not introduced by the transformation. This suggests that a property-preserving transformation would transform correct CIF controller models to correct PLC programs.

Vanderlande employs a maximum cycle time for a PLC controlling 200 sections. It is the eventual goal of the SUCCESS project to generate software that meets this requirement. During this project, it was not met. However, the performance was improved compared to that of the previous project, despite adding extra functionality to both the CIF controllers and the transformation. Furthermore, efficiency improvements are still in development and more options for possible improvements are yet unexplored.
6.2 Recommendations

In this project, its predecessors and a project currently in development, model definitions and functionality has been implemented in all LLC control levels including communication to higher and lower system levels. This does not mean that the control library is finished, but that the major challenges are no longer in the control library. However, there is still much room for small improvements in the existing models. In this project the focus was on reevaluating and improving the existing control definitions. The hardware models were often not be reviewed as thoroughly, because the controller models had priority. For instance, the hardware model for the conveyor can still only handle 3 products at a time. The methodology applied in Chapter 4 would be very useful for finding such areas for improvement. Supporting more types of objects in the hardware models is important for testing larger BHS systems. Such models should be configurable in their output. For instance, a label scanner should be able to fail a measurement and give a no read, or multiple label numbers.

New projects could focus on developing controllers for equipment which is normally not standardized in the library, such as very project-specific sorting zones. These kinds of projects would lead to further research into the direct industrial application of CIF generated control software. Furthermore, it would allow for a more in-detail comparison between normal and generated PLC software. Is CIF modeling faster than direct PLC programming for operating such systems?

Until now, CIF controllers have only been applied on systems completely controlled by CIF generated software. However, industrial application will, especially in the beginning, be alongside software developed in more conventional ways. Modeling an independently operated system and implementing generated control software alongside normal PLC programs should give an indication of how well the two approaches can be integrated. Due to the differences between normal and generated PLC software, it is probably not practical to run both on one PLC.

A new challenge is automatic generation of CIF code. As the specifics of large control systems are often extensively documented in structured formats, there is no reason to do all instantiation manually. Automatically generating a controller model, a hardware model, and the I/O mapping for the PLC from such a generic specification would save time and eliminate a source of errors. Before this step is taken, the way definitions are parameterized should be reviewed. Once an automatic code generation has been established, making large changes to the parameter interfaces becomes difficult and requires more work. Moreover, the input parameters could probably be optimized and bundled. For instance, the use of custom tuples/types in CIF and User Defined Types in PLC has now been thoroughly tested in the transformation. Input parameters of instantiations could be more intuitively ordered in tuples, types and enumerations. Once a generic and versatile interface is created, automatic generation of CIF models can be considered.
The possibilities for automatic testing or run-time verification of control requirements in CIF should be researched. ToolDef is a language developed by the Systems Engineering Group for controlling and compiling their developed tools and was recently updated to ToolDef-2.0.0. The CIF simulator in combination with the recently extended ToolDef environment, creates new opportunities for automatically and sequentially running test cases. Using automata operating alongside the simulated tests, it can be proven that in all the scenarios tested, specific requirements were met. These tests would not need a visualization, making it possible to simulate much faster than real-time. This type of testing does not prove that there is no possible path in which the requirement is not met. However, by using different scenarios and randomly generated starting parameters, it can be shown to be correct in a large number of scenarios.

For the implementation of automata detecting requirement failure or success, a number of important topics can be defined:

- After each test case, the automaton should indicate whether the requirement held or not during the tests. A boolean variable turning true or false or a location deadlocking the automaton are examples.

- An overview of the different types of requirements and what they require for validation must be created. If input, output and hardware status are sufficient, no access to the internal variables of the controllers would be required. Probably, a distinction can be made between variables that can be validated using only the hardware models and requirements for which the automata depend on the actual implementation of the requirement.

- Based on the information needed to determine requirement failure, a testing interface can be determined. Preferably, the implementation of run-time verification should not influence the control library. During simulation, the controller and hardware models are merged. A set of verification automata could also be included for automatic testing, possibly by merging in a third file. By using the input and output keywords, the required information can be exchanged with the controller and hardware models.

- In order to diagnose and solve problems detected with run-time verification, information on the specific failure of the requirements must be saved. For instance, the states of the system and the order of events could be logged. Perhaps the automata could be designed to give some indication towards a probable cause.

- The process of defining, updating and maintaining these automata needs to be integrated in the design process of CIF controllers. Preferably, the automata can be defined by everyone with a list of requirements, and no complex knowledge of the controller model is required. Otherwise, during design of new controller models, the testing automata need to be defined as well. Furthermore, the automata should be versatile enough to be easily adjusted when the requirement or the controller/hardware models are altered.
Although run-time verification can reduce the need for interactive visualization, supervised tests will still be required. For these reasons the efficiency of the interactive visualization should be improved. For small models, the interactive visualization is very useful. However, in order to make it easier to test larger models, improvements are required. Issues are the speed of the visualization, the lack of dynamic scaling of the generated images and dynamic access to variables during simulation.

In commissioning a control system, invariably the actual hardware differs from the specifications used in the controller design. As a result, changes have to be made to the control system. The generated PLC program can be altered to change system parameters like conveyor length and PEC location. However, for more complex changes, a new program has to be generated from the CIF source model. In order to quickly make such changes, the road from CIF model to implemented software on the PLC should be shorter. The transformation itself takes only minutes, and importing and compiling the SCL code in TIA portal is the real bottleneck. The time required to generate and compile the blocks from the SCL source fluctuates and can in some cases take up to 30 minutes. Only when these steps are completed can the program be downloaded to the actual PLC. While both version 11 and 13 of TIA Portal were used in this project, no significant improvement to the importing mechanism was observed. It is likely that the importing mechanism was never meant to be used repeatedly or on the scale of an entire PLC program. The road from CIF program to a new version of the program on the PLC needs to be more efficient, either by improving TIA portal or by finding an alternative solution for creating the PLC program blocks from the generated SCL source.

Research could be done on the possible implementation of the modeling procedures and tools of this project in the Vanderlande design process. Vanderlande has worked with PLC control for decades, which is reflected in the design cycle and the skill sets of their employees. Experiments could look into how well PLC engineers can adapt to a CIF way of modeling. Such tests should give an indication of the possible problems and the willingness and ability of employees to change their way of working. It is important that the advantages of the implementation of CIF modeling are properly highlighted.

As illustrated in this section, there are still many possibilities for further research. The CIF control model framework can still be extended in both the definition library and the surrounding functionality, creating a more complete software environment capable of generating correct and validated software for different software platforms.
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BHS</td>
<td>Baggage Handling Systems</td>
</tr>
<tr>
<td>BR</td>
<td>Baggage Record</td>
</tr>
<tr>
<td>BSO</td>
<td>Baggage Scanning Object</td>
</tr>
<tr>
<td>bw</td>
<td>Backward</td>
</tr>
<tr>
<td>CIF</td>
<td>Compositional Interchange Format</td>
</tr>
<tr>
<td>CIZ</td>
<td>Check-In Zone</td>
</tr>
<tr>
<td>DB</td>
<td>Data Block</td>
</tr>
<tr>
<td>DES</td>
<td>Discrete Event System</td>
</tr>
<tr>
<td>FB</td>
<td>Function Block</td>
</tr>
<tr>
<td>fw</td>
<td>Forward</td>
</tr>
<tr>
<td>HBO</td>
<td>Hold Baggage Screening Object</td>
</tr>
<tr>
<td>HBZ</td>
<td>Hold Baggage Screening Zone</td>
</tr>
<tr>
<td>HLC</td>
<td>High Level Controllers</td>
</tr>
<tr>
<td>IDZ</td>
<td>Identification Zone</td>
</tr>
<tr>
<td>LLC</td>
<td>Low Level Controllers</td>
</tr>
<tr>
<td>LMS</td>
<td>Local Motor Starter</td>
</tr>
<tr>
<td>MTZ</td>
<td>Carrousel</td>
</tr>
<tr>
<td>OB</td>
<td>Organisation Block</td>
</tr>
<tr>
<td>PEC</td>
<td>Photo Electric Cell</td>
</tr>
<tr>
<td>PLC</td>
<td>Programmable Logic Controller</td>
</tr>
<tr>
<td>POU</td>
<td>Program Organisation Unit</td>
</tr>
<tr>
<td>SCL</td>
<td>Structured Control Language</td>
</tr>
<tr>
<td>SCT</td>
<td>Supervisory Control Theory</td>
</tr>
<tr>
<td>SUCCESS</td>
<td>Supervisory Control of Concurrent Embedded Systems</td>
</tr>
<tr>
<td>SVG</td>
<td>Scalable Vector Graphics</td>
</tr>
<tr>
<td>SWS</td>
<td>Switch Section</td>
</tr>
<tr>
<td>TIA</td>
<td>TIA portal 13</td>
</tr>
<tr>
<td>TRS</td>
<td>Transport Section</td>
</tr>
<tr>
<td>TRZ</td>
<td>Transport Zone</td>
</tr>
<tr>
<td>UFO</td>
<td>Unknown Product</td>
</tr>
<tr>
<td>VB</td>
<td>Vertibelt</td>
</tr>
<tr>
<td>VBZ</td>
<td>Vertibelt Zone</td>
</tr>
</tbody>
</table>
Bibliography


Appendix A

Test Phase (Confidential)
Appendix B

Vanderlande Terms and Sources (Confidential)
Appendix C

The BR Tuple (Confidential)