Model-based design of baggage handling systems

Model-based design of baggage handling systems

Lennart Swartjes

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-4548-3
IPA Dissertation Series: 2018-12

Work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics)

Model-based design of baggage handling systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op donderdag 13 september 2018 om 13:30 uur

door

Lennart Swartjes

geboren te Utrecht

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr. L.P.H. de Goey
promotor:	prof.dr. W.J. Fokkink
copromotoren:	dr.ir. D.A. van Beek
	dr.ir. M.A. Reniers
leden:	prof.dr. T. Villa (Università di Verona)
	prof.dr. J.J.M. Hooman (Radboud Universiteit)
	prof.dr.ir. J.F. Groote
adviseur:	dr.ir. J.A.W.M. van Eekelen (Vanderlande Industries)

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Preface

This thesis marks the final milestone of my Ph.D. project. The project was executed at the Eindhoven University of Technology in cooperation with Vanderlande Industries. One of the benefits of working together with industry is the ability to apply most of my findings to a real-life system or case. The combination of academics and industry made my project very interesting and I would like to thank Vanderlande, and in special Gert Bossink, Joost van Eekelen, and Vincent Kwaks, for enabling this cooperation.

I would like to thank all the persons who contributed to my Ph.D. project. I would like to thank Bert van Beek, Wan Fokkink, and Michel Reniers for all their time spent guiding me and providing helpful advice. I learned some valuable lessons on project management and conveying my results. From Vanderlande, I would like to thank Joep Bax, Joost van Eekelen, René Hommels, Fred Verstraaten, and many others for their feedback and input during our many meetings. All have helped to create the result as it is today.

I also would like to thank all committee members for their time reviewing this thesis and providing valuable feedback. Although it took longer than usual, the feedback and the resulting adaptations helped to convey the message in a more structured and comprehensible form.

Thanks goes out to all students that contributed to the thesis (Rik Kamphuis, Sjors Jansen, Tom Zwijgers, and Oriane Dierikx) and to Dennis Hendriks and Albert Hofkamp for supporting the CIF3 tooling.

I would like to thank my parents, Leendert and Yvonne Swartjes, and my sister, Lysanne Swartjes, for being supportive during the times I was lost and did not know how to proceed. I would like to also thank them for being genuinely interested in what I was doing. The same holds for my best friend, Rozemarijn Missler, with whom I have shared many coffees on the different subjects of my Ph.D. project. In addition, I would like to thank Rozemarijn and Lysanne for being my seconds during my defence.

Finally, I would like to thank all my Nieuwegein friends, Eindhoven friends, and friends I have made during my Ph.D. project for their support. I enjoyed all the coffee moments, crossword moments, parties, and (in-depth) conversations. All these events have contributed to the results of this project, in one way or another. Special thanks go out to René Kouwer for designing the cover of this thesis.

Summary

A baggage handling system (BHS) is a key component of an airport. A BHS is responsible for transporting baggage items from designated entry points, e.g., check-ins, to designated exits, e.g., airplanes. To this end, a BHS consists of many kilometers of conveyor belts and specialized components for the routing of baggage items. To control these systems, a supervisory controller is used. Such a controller determines one or more control actions to perform, based on the observable and reconstructed state of the system. Observing the state is done by means of sensors, e.g., photo cells, while the actions are performed by the actuators of the system, e.g., motors. Currently, these supervisory controllers are created based on a set of requirements. From these requirements, the code of the controller is manually written and finally tested on (a prototype of) the real system. Using this traditional approach, it is very difficult to find the source of errors. As a result, lengthy design cycles are needed before a controller is accepted.

Four observations are made why the traditional approach is lengthy for BHSs. First, conformance of the controller with respect to the requirements cannot be assessed before the controller is coded. Second, there are many client-specific requirements for BHSs, leading to a lot of rework. Third, a BHS has a high level of complexity as the many components are inter-connected. Last, both geographical requirements, e.g., stopping products at specific points, and temporal requirements, e.g., a minimum distance between any two products, must be implemented using information from relatively few sensors.

A known solution to cope with these difficulties, is to use a model-based design approach with early integration. Using model-based design, a model is made of the controller (requirements) instead of directly coding the controller. The idea behind early integration is to test earlier, before the complete controller is available. To this end, a model of the uncontrolled system is created. Simulating the model of the uncontrolled system together with the controller model, the behavior can be validated. Code generation is used when the model is accepted to reduce the number of coding mistakes and to abstract from the required control platform. An alternative to manually modeling the controller is synthesis. Given a model of the control requirements, a controller may be synthesized. The benefit is that this synthesized controller satifies the requirements for any given scenario.

The goal of the Ph.D. project is to investigate whether model-based design of a supervisory controller for an industrial-sized baggage handling system is feasible, from design to real-time PLC implementation. Based on this overall goal, four challenges are identified:

(1) How to exploit MBD and early integration for supervisory controllers (for BHSs) to achieve models with an increased quality and flexibility and a reduced complexity?
(2) How to provide user-feedback on the removal of states during synthesis in order to increase the usability of supervisory control synthesis in industry?
(3) How to perform model-to-model transformations, while retaining full traceability information on the requirements?
(4) How to generate code from models such that the execution of these models on a real-time platform is similar to the validated behavior?

In Chapter 2, the creation of models is explained. Automata, a way of modeling the discrete states of the system and the transitions between those states, are used extensively. Because there are several ways of modeling the same system, modeling considerations are defined that lead to an increased quality and flexibility and reduced complexity for BHSs. Where appropriate, these considerations are generalized for arbitrary models. To support early integration, different validation techniques are shown that can be used to assess the quality of the model before the controller is coded or before the actual system is built.

In Chapter 3, the synthesis of supervisory controllers is considered. It is observed that there are a couple of reasons why supervisory controller synthesis is not widely adopted in industry. One of these reasons is the absence of user-feedback when desired behavior is excluded for the supervised system. To provide user-feedback, a sound and complete deduction system is introduced for the derivation of causes on the absence of specific states from the supervised system. Using this deduction system, an algorithm is defined that deduces a single cause for each absent state in the supervised system during synthesis.

In Chapter 4 and 5, it is investigated how traceability can be preserved for model-to-model transformations. A specific model-to-model transformation, i.e., the removal of synchronous behavior, is taken as an example. First, a formal definition of traceability is defined which, in essence, relates the syntax of two models. Second, an alternative algorithm for the removal of synchronous behavior is introduced that better suits code generation. For this algorithm it is proven that the result is traceable.

In Chapter 6, the real-time implementation of BHSs is considered. First, common mismatches between the used modeling framework and the control platform are investigated and solved. Second, a code generator is created that works specifically for BHS models and their requirements on the resulting code. Third, the code generator is used to solve three industrial-sized cases. The first case comprised the control of a real-life system, consisting roughly of 45 components. The correctness of the model is shown by testing on the system. The second and third case comprised the control of a real airport. In the second and third case emulation is used to show the correctness of the model.

In conclusion, a framework is introduced in which BHSs can be modeled with increased quality and flexibility and reduced complexity while early integration can be exploited to validate the models before the controller is coded or an actual system is built. A framework is created to aid users in gaining insights on the exclusion of desired behavior during synthesis. A formal definition of traceability is introduced. An algorithm is introduced for the removal of synchronous behavior for which the resulting model is traceable with respect to the original model. Finally, it is shown how these models can be used for real-time implementation by showcasing three large industrial-size cases for which working code has been created.

Samenvatting

Een bagageafhandelingssysteem (BAS) is een uitermate belangrijk onderdeel van een luchthaven. Een BAS is verantwoordelijk voor het transporteren van bagagestukken vanaf een specifiek ingang, bijvoorbeeld een check-in, naar een specifieke uitgang, bijvoorbeeld het vliegtuig. Om dit mogelijk te maken bestaat een BAS uit vele kilometers lopende band en specifieke componenten om de bagagestukken te sturen. Om een BAS aan te sturen wordt een besturing gebruikt welke op basis van de geobserveerde en gereconstrueerde toestand van het systeem bepaalt welke acties het systeem mag uitvoeren. Het observeren van het systeem gebeurt meestal met sensoren, zoals een fotocel, terwijl de acties van het systeem worden uitgevoerd door actuatoren, zoals motoren. Op dit moment worden deze besturingen geprogrammeerd door software-ontwikkelaars aan de hand van een set van eisen. Deze code wordt vervolgens getest op (een prototype van) het systeem. Omdat deze code direct wordt geprogrammeerd is het vaak moeilijk te achterhalen waardoor en waarom een fout optreedt. Daarom kost het veel tijd voordat de besturing geaccepteerd wordt.

Vier redenen kunnen worden aangemerkt waarom deze traditionele manier van werken veel tijd kost. Ten eerste, er kan pas getest worden nadat de besturing geïmplementeerd is. Ten tweede, een BAS heeft veel klant-specifieke eisen, waardoor elke BAS opnieuw werk vergt. Ten derde, een BAS is zeer complex omdat de verschillende componenten in sterke mate van elkaar afhankelijk zijn. Als laatste, zowel geografische eisen, zoals het stoppen van bagagestukken op een specifieke plek, als temporele eisen, zoals een minimale afstand tussen bagagestukken, moeten worden geïmplementeerd gebruik makende van relatief weinig sensoren.

Een mogelijkheid om deze problemen te voorkomen is om modelgebaseerd te ontwerpen en gebruik te maken van versnelde integratie. Met een modelgebaseerde aanpak wordt een model gemaakt van de (eisen van) de besturing in plaats van het direct programmeren. Het idee van versnelde integratie is dat dit model getest kan worden voordat de besturing volledig is geïmplementeerd. Om dit voor elkaar te krijgen wordt er een model gemaakt van het niet-bestuurde systeem, welke wordt gecombineerd met het model van de besturing om het gedrag van het bestuurde systeem te kunnen valideren. Automatische generatie van de code van de besturing, nadat het gedrag is goedgekeurd, zorgt ervoor dat de code geen programmeerfouten bevat en onafhankelijk is van het besturingsplatform. Een alternatief is het automatisch creëren van het model van de besturing door middel van synthese. Synthese maakt gebruik van een model van de eisen. Een bijkomend voordeel is dat een gesynthetiseerde besturing onder alle gevallen voldoet aan de eisen.

Het doel van dit Ph.D. project is om te onderzoeken of een modelgebaseerde aanpak toepasbaar is voor het hele ontwikkeltraject van besturingen van BASen, vanaf het ontwerp tot de implementatie van de besturing op een PLC. Aan de hand van deze vraag zijn er vier onderzoekspunten vastgesteld:

(1) Hoe kan een modelgebaseerde aanpak en versnelde integratie worden gebruikt om modellen met een verhoogde kwaliteit en flexibiliteit en een verlaagde complexiteit te verkrijgen?
(2) Hoe kan een eindgebruiker geïnformeerd worden over welke toestanden er worden verwijderd tijdens synthese, om synthese meer toepasbaar te maken in de praktijk?
(3) Hoe kunnen modeltransformaties worden uitgevoerd zonder verlies van traceerbaarheid van de eisen?
(4) Hoe kan code gegeneerd worden uit de modellen zodat de executie van deze code gelijk is aan het gevalideerde modelgedrag?

In hoofdstuk 2 wordt het maken van modellen door middel van automaten uitgelegd. Automaten worden gebruikt om de discrete toestanden en de transities tussen deze toestanden te modelleren. Omdat er verschillende manieren zijn om hetzelfde systeem te modelleren, worden er in dit hoofdstuk verschillende modelleeroverwegingen geïntroduceerd die leiden naar modellen van BASen met een verhoogde kwaliteit en flexibiliteit en een gereduceerde complexiteit. Wanneer dit kan worden deze overwegingen gegeneraliseerd. Om versnelde integratie mogelijk te maken worden verschillende validatietechnieken getoond die kunnen worden gebruikt om de kwaliteit van het model te bepalen voordat de besturing daadwerkelijk geprogrammeerd is en het systeem gebouwd is.

Hoofdstuk 3 is gewijd aan de synthese van besturingen. Er wordt opgemerkt dat synthese van besturingen nog geen gemeen goed is in de industrie. Een van de redenen is de afwezigheid van terugkoppeling aan eindgebruikers; wanneer gewenst gedrag wordt geblokkeerd door de besturing is er geen mechanisme om te achterhalen waarom dit zo is. Om deze terugkoppeling te kunnen maken is een correct en compleet deductiesysteem gemaakt dat de reden van afwezigheid van een bepaalde toestand kan verklaren. Gebruik makende van dit deductiesysteem wordt het synthese-algoritme uitgebreid zodat deze 1 reden kan teruggeven voor elke toestand die door de besturing wordt geblokkeerd.

Hoofdstuk 4 en 5 bestuderen traceerbaarheid wanneer er model-naar-model transformaties worden uitgevoerd. De focus ligt op een specifieke transformatie, namelijk het elimineren van synchroon gedrag. Een formele definitie van traceerbaarheid wordt geïntroduceerd in hoofdstuk 4. Deze traceerbaarheidsrelatie relateert de syntax van modellen met elkaar. Een alternatief algoritme voor de eliminatie van synchroon gedrag wordt gedefinieerd in hoofdstuk 5. Voor dit algoritme wordt bewezen dat de traceerbaarheid niet wordt aangetast.

In hoofdstuk 6 wordt de implementatie van BASen op een real-time besturingsplatform besproken. Ten eerste worden de algemene verschillen tussen het modelleerplatform en het besturingsplatform besproken en vervolgens opgelost. Een programmagenerator wordt daarna geïntroduceerd welke specifiek gemaakt is voor BASen en hun specifieke eisen. Deze generator is gebruikt om drie industriële cases op te lossen.

De eerste case betreft het maken van een besturing voor een echt systeem, bestaande uit ongeveer 45 componenten. De correctheid van de besturing is aangetoond door testen op dit systeem uit te voeren. De tweede en derde case betreffen het aansturen van een bestaande luchthaven. Voor deze cases is gebruik gemaakt van emulatie om de correctheid aan te tonen.

In conclusie, een methodiek voor het modelleren van BASen is geïntroduceerd welke het mogelijk maakt de kwaliteit en de flexibiliteit van de modellen te verhogen en de complexiteit te verlagen, terwijl versnelde integratie gebruikt kan worden om het gedrag te valideren voordat de besturing is geprogrammeerd of het systeem gebouwd is. Bovendien is voor synthese een methodiek geïntroduceerd welke het mogelijk maakt redenen af te leiden voor de afwezigheid van gewenst gedrag. Een traceerbaarheidsrelatie en een alternatief algoritme voor het elimineren van synchronisatie zijn gedefinieerd. Als laatste is getoond hoe deze modellen geïmplementeerd kunnen worden en is de haalbaarheid aangetoond door programma's van besturingen te creëren voor drie grote industriële cases.

Contents

Chapter 1

Introduction

In this chapter, the concept of baggage handling systems is introduced. Based on observations made at Vanderlande Industries on the current way of creating supervisory controllers, challenges are defined that will be investigated further in this thesis. This chapter is partly based on Swartjes et al. (2017b).

1.1 Baggage handling systems

Airports use dedicated systems to transport baggage items from designated entry points, e.g., a check-in, to designated exit points, e.g., an aircraft. These systems are commonly referred to as baggage handling systems (BHSs). Typically, a BHS consists of many kilometers of conveyor belts to transport baggage items, resulting in systems as shown in Figure 1.1. In addition, it contains many specialized components to screen and (re)route items.

One supplier of BHSs is Vanderlande Industries. The following introduction is taken directly from their website (http://www.vanderlande.com/):

> *Vanderlande is the global market leader for value-added logistic process automation at airports, and in the parcel market. The company is also a leading supplier of process automation solutions for warehouses. Vanderlande's BHSs move 3.7 billion pieces of luggage around the world per year, in other words 10.1 million per day. Its systems are active in 600 airports including 13 of the world's top 20.*

BHSs are built up in a hierarchical way according to Vanderlande's specifications. In this thesis, only sections, zones, and areas are considered. A section is a standardized part or component of the system. Standardization in this context means that, excluding some exceptional cases, each section in any system is mechanically as well as electronically equal. A section fulfills one particular task, like the transportation of products. Such a section, responsible solely for the transportation of products, is called a TRansport Section (TRS) and is depicted in Figure 1.2. A TRS consists of

Figure 1.1: Partial overview of a BHS (©Vanderlande Industries).

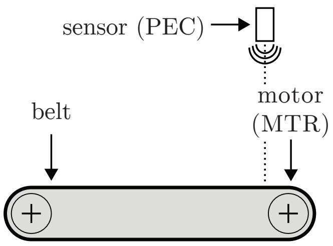


Figure 1.2: Schematic overview of a TRansport Section (TRS). The TRS contains a conveyor belt, a sensor to detect products, and a motor.

a conveyor belt, a motor that drives the belt, and a sensor to detect the presence of a product directly underneath it. Note that the latter implies that only the change of the absence to the presence of a baggage item, and vice versa, is detected by the sensor.

Multiple sections together create a zone. Again, a zone is standardized and fulfills a particular task. The most simple zone, consisting of solely TRSs, is a TRansport Zone (TRZ). Although a zone is standardized, a TRZ may contain any number of TRSs.

As there are many routes in a BHS that may lead to the same destination, a BHS contains numerous routing points. Specialized zones like the VertiSorter Zone (SVZ), depicted in Figure 1.3, sort products vertically. Products enter from the left and are sorted out to either the top or the bottom TRS. A specialized section called a SWitch Section (SWS) is present to change the sorting direction. Hence, this specific zone consists of three TRSs and a SWS.

The final level is called an area, consisting of multiple zones. An area is always tailored to the customer's needs. An example of an area is an ARivals Area (ARA) which contains all the zones necessary for reclaiming products. In conclusion, an area

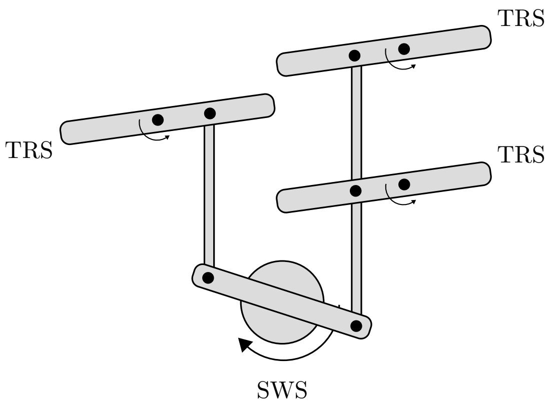


Figure 1.3: Schematic overview of a VertiSorter Zone (SVZ) consisting of three TRSs and a SWS.

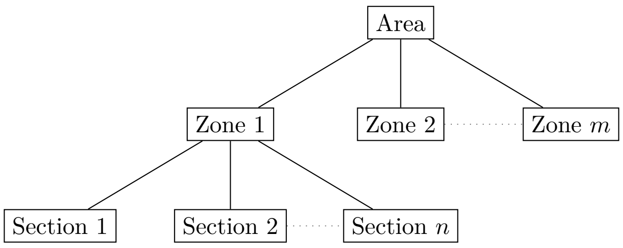


Figure 1.4: The hierarchical structure of an area, containing multiple zones and sections.

has a tree-like structure as depicted in Figure 1.4.

1.2 Supervisory controllers

At the heart of such a BHS is a supervisory controller. The controller reads the observable state of the system via the current values of the sensor signals. Based on this state, and the controllers internal state, it calculates the new internal state and the required values of the actuators. Note that supervisory controllers mainly deal with logic control and sequence control, and are often represented as finite state automata (Cassandras and Lafortune, 2009). A schematic overview of a system under supervision is depicted in Figure 1.5.

1.3 Control development process

The current approach to the design and creation of controllers for BHSs can be captured by the well-known V-model (Pressman, 2005). First, the requirements of the system are defined: what should the system do, and how should it complete this task.

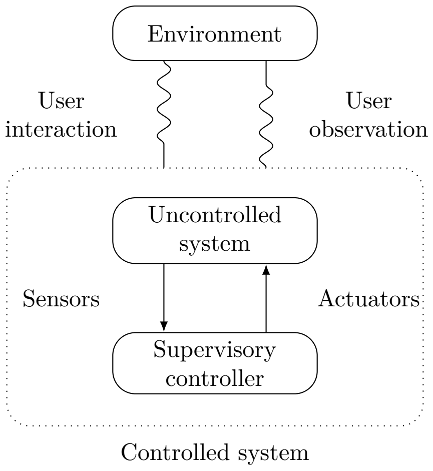


Figure 1.5: Schematic representation of a system under supervision.

From these (behavioral) requirements, the specifications of the system are defined by the system architect and written down in a design document. Next, the software engineer encodes the supervisory controller based on the design document. Rigorous testing is employed during development and commissioning of the system. If all tests succeed, the controller is accepted.

In general, errors are found during tests. To resolve these errors, the cause of the error must be found. However, in many cases, finding the cause is difficult. As the controller is based on the software engineer's interpretation of the design document, the specification may be wrong, the specification may have been wrongly interpreted, or the controller may contain a coding error. Therefore, many lengthy design cycles are needed before a controller is accepted.

It may even occur that requirements are forgotten, i.e., not implemented, or that errors remain unnoticed during tests. This may lead to disastrous results when the system goes into operation. A well-known example is Denver International Airport (De Neufville, 1994; Jackson, 2006), where a faulty (software) design resulted in large delays and huge financial losses. More recently, in June 2017, the BHS at Heathrow Airport suffered a failure, causing many passengers to fly without their luggage. This happened only three weeks after a catastrophic IT system failure at British Airways caused a full day cancellation of all flights from Heathrow and Gatwick.

1.4 Observations

To reduce the occurrence of errors, the process can be partly automated by means of standardization (Tassey, 2000): standardized components can be associated with standardized code fragments with proven functionality and quality. Nevertheless, creating a proper controller remains challenging, and requires a lot of effort. The reason for this is fourfold:

First, conformance of the controller with respect to the design requirements cannot be assessed before an actual implementation of the controller is built. When there is an implementation, testing can take place by means of, e.g., emulation. In other words, the quality (IEEE, 2014; Wagner, 2013) of the controller design remains unknown until it is actually built.

Second, there are many client-specific requirements, as each BHS is tailored to a specific airport. The implementation of these client-specific requirements induces a lot of additional work, as these components are not standardized. As a result, code fragments influenced by these requirements must be (re)created from scratch. Moreover, client- or government-specific needs may also induce additional requirements; legislation may govern the use of a specific control platform. This entails that, even for standardized components, additional work is needed to make these components compliant with the country-specific control platform. In other words, there is little to no flexibility (IEEE, 2010; Eden and Mens, 2006) in the way controllers are designed currently.

Third, a BHS contains many components that, in some way or another, are connected with one another. Because of this large number of interactions, it is difficult to oversee all design choice implications. As a result, a BHS has a high level of complexity.

Last, a serious challenge in the design of supervisor controllers for BHSs is that both geographical requirements, e.g., stopping products at specific points, and temporal requirements, e.g., a minimum distance between any two products, must be implemented using information from relatively few sensors. For instance, a single photo-electric cell is used to control a single belt that may contain many baggage items. Based on the discrete information from this sensor, i.e., the presence or absence of a product, both the geographical and temporal requirements of that belt have to be fulfilled.

1.5 Model-based design

With the ever increasing complexity of systems, and therewith BHSs, the complexity of the associated supervisory controllers also increases. As a consequence, the time to develop new products or add/change functionalities increases, leading to higher costs and a reduced competitive advantage. Therefore, it is desired to reduce the effort of creating supervisory controllers, i.e., the time it takes from specification to acceptance, while simultaneously increasing:

Quality	Compliance of the controller with respect to the design document (IEEE, 2014).
Flexibility	Adaptiveness (or ease of adaptation) of the controller (IEEE, 2010).

In short, it is desired to reduce the number and length of design cycles needed to create a supervisory controller (for BHSs).

A well-known solution to achieve the latter is by using a **model-based design** (MBD) and by allowing **early integration** (Braspenning, 2008). In an MBD formalism, a model is made of the controller (requirements) instead of directly coding the controller. When models are used with a fixed semantics, both the system architect and the software engineer can interpret the controller in the same way, reducing the number of interpretation errors.

The idea of early integration is to test earlier, before the complete controller is available. For instance, compliance of each component with respect to its specification can be tested separately. In such a way, errors are found earlier in the design process; a full redesign of a system is a known risk when finding (major) errors late in the design process.

An alternative method to manually specifying the supervisory controller is supervisory controller synthesis. Instead of directly modeling the resulting controller, a model of the uncontrolled system and a model of the requirements are made. For the sake of synthesis, some control actions of the uncontrolled system are classified controllable and some uncontrollable. The supervisor may only control the controllable actions of the uncontrolled system. Typically, uncontrollable events are associated with events from sensors, e.g., the detection of a product, while controllable events are associated with events of actuators, e.g., starting a motor.

The goal of synthesis is to create a supervisor such that the controlled system, i.e., the supervisor applied to the uncontrolled system, abides to the specifications. In other words, the result of the supervisor is a controlled system which does not exhibit unwanted behavior, never deadlocks, i.e., can always exhibit some desired behavior, does not intervene with the uncontrollable part of the system, and intervenes as little as possible with the controllable part of the system. If the controlled system abides to the latter four properties, the system is said to be, respectively, safe, nonblocking, controllable, and maximally permissive. Note that (additional) liveness conditions may be taken into account, e.g., time optimal supervisors (Swartjes et al., 2011). However, this is not considered in this thesis.

Finally, code generation is used to reduce the number of coding mistakes and to abstract from the required control platform. This increases the quality of the controller code, while simultaneously increasing the flexibility, as code can be generated from one model for multiple control-platforms.

In Figure 1.6, an overview of the framework is given, which aims to reduce the effort of creating supervisory controllers while increasing quality and flexibility. The idea is that, given a model of the controller, a model of the hardware, and (potentially) a model of the environment, validation, verification, code generation, and emulation, is possible within a single MBD environment. The controller model can either be modeled, or can be synthesized based on models of the uncontrolled system, i.e., the plant, and the requirements.

1.6 CIF

CIF (`http://cif.se.wtb.tue.nl/`), and its associated toolset, are used extensively in our approach. CIF is used for the creation and validation of models, as well as the generation of code from these models for different control platforms.

The Compositional Interchange Format (CIF) came to life as a solution to the many different modeling formalisms in existence. CIF provides an intermediate rep-

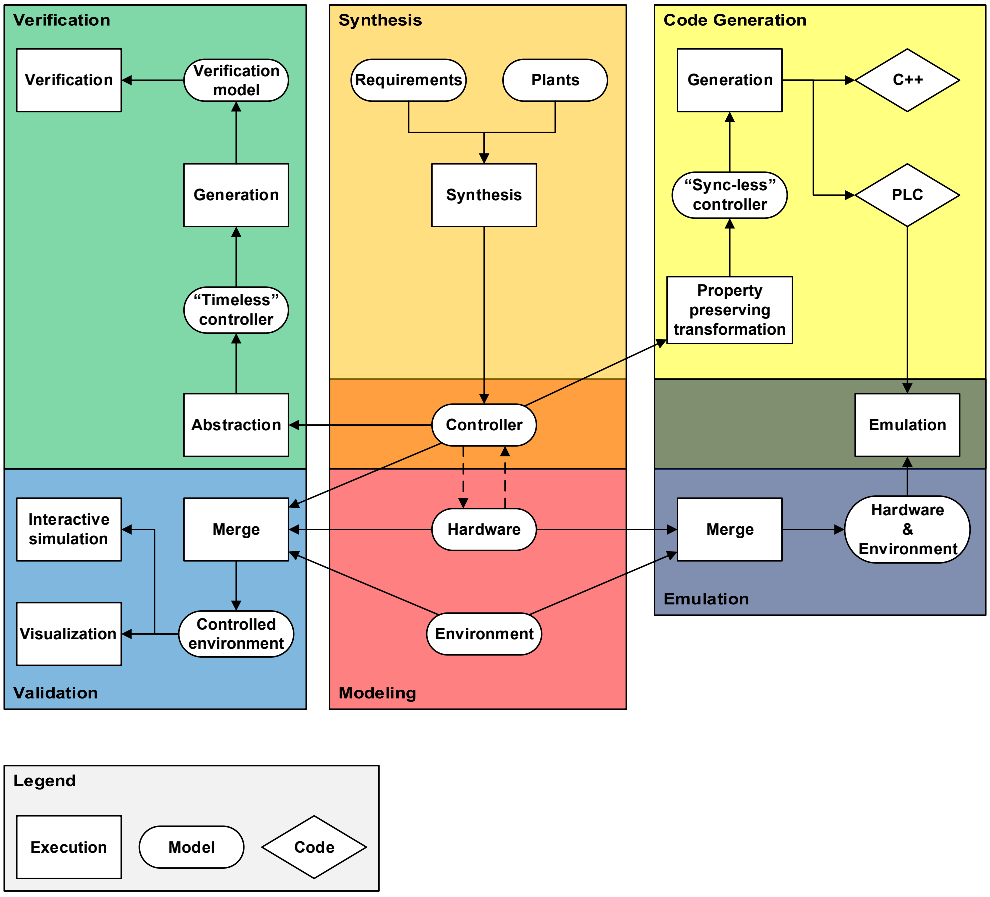


Figure 1.6: Overview of the framework used during the Ph.D. project.

resentation of models and many transformations to and from this representation. Using these transformations and the intermediate representation, models can be easily transformed from one formalism to another. This allows for tight integration of different tools in the design process; it is possible to validate, verify, and more, by transforming a single model to the appropriate representation of those specific tools.

Nowadays, CIF has evolved to a stand-alone MBD framework that supports the design of supervisory controllers from specification to acceptance (van Beek et al., 2014). CIF models are specified textually in the formalism of hybrid automata (Nadales Agut et al., 2013). This syntax is parsed on-the-fly and errors in the specification are directly reported to the end-user by means of different visual cues. CIF provides internal support for supervisory controller synthesis, validation by means of interactive simulation and visualization, and the generation of controller code. CIF still provides several model transformations to allow integration of external tools. For instance, transformations are present to verification tools such as UPPAAL (Bengtsson et al., 1996) and mCRL2 (Groote et al., 2007).

1.7 Challenges

The overall challenge that was at the heart of this Ph.D. project is stated as follows:

> Investigate whether model-based design of a supervisory controller for an industrial-sized baggage handling system is feasible, from design to real-time PLC implementation.

In this context, two different design methodologies need to be compared. The overarching research question above was divided into four more specific challenges, for which it can be analyzed how they influence and possibly improve the design process. Combining the observations made on creating supervisory controllers for BHSs and the usage of an MBD approach as sketched in the previous section, four challenges have been identified that will be addressed in this thesis. The first challenge is the most straightforward one, directly related to MBD of supervisory controllers.

> How to exploit MBD and early integration for supervisory controllers (for BHSs) to achieve models with an increased quality and flexibility and a reduced complexity?

In addition to manually creating a supervisory controller, a supervisor can be synthesized. Safeness, nonblockingness, and controllability are key features why supervisory control synthesis is an anticipated concept in industry. However, adoption of supervisory control synthesis is still low: only a handful of examples can be found on the applications of supervisory control synthesis in industry, e.g., Vahidi et al. (2006); Reijnen et al. (2017).

Two main reasons can be pinpointed for this low adoption. First, scaling is an issue. For industrial-sized systems, both memory problems and long computation times may be encountered, although recently improvements have been made to the algorithms to improve scalability, e.g., Miremadi and Lennartson (2016).

Second, a deficiency is the lack of user-feedback. During the early stages of the supervisory controller design, desired behavior may be unexpectedly excluded by the supervisor or, even worse, the synthesis algorithm frequently returns an empty supervisor. User-feedback is not provided that explains why a supervised system could not be synthesized, or why certain behavior is absent. Finding reasons demands knowledge on the synthesis algorithm, which is normally nonexistent in industry. Moreover, it requires working out a long chain of steps in which states are successively excluded, which may be too demanding even for experts.

> How to provide user-feedback on the removal of states during synthesis in order to increase the usability of supervisory control synthesis in industry?

Looking at Figure 1.6, a model may be used for different kinds of purposes: validation, verification, and code generation. However, a single model may not fit all the formalisms needed to allow validation, verification, and/or code generation. As an example, verification tools often provide a specific modeling formalism which must be used to model the system to be verified. Therefore, in order to use this verification tool, the model must be transformed to a model using this formalism while retaining the same behavior. This can be done either manually of automatically.

The goal of validation and verification is to assess the correctness of the model with respect to the requirements. Therefore, it is of importance that the modeler can still derive which requirement is responsible for which parts of the system, even after transforming the model from one modeling formalism to another. Otherwise, the requirement responsible for the erroneous behavior cannot be traced back from the results coming for the verification tooling. A similar observation may be made for code generation, for which it may be desired that it is known which code fragments are related to which requirements.

How to perform model-to-model transformations, while retaining full traceability information on the requirements?

To reduce the effort of building the actual controller, it is desired to directly generate code from the controller model. This has the benefit that the number of coding errors is reduced. Additionally, a major benefit for the industry is that one model can be used for the generation of code for different platforms.

Major differences between CIF and general control platforms include the progression of time, i.e., the difference in model time and real-world time, the lack of synchronization, and the update semantics. More details are provided in Chapter 6. As the executional semantics of CIF and an arbitrary control platform differ, these differences need to be taken into account. Otherwise, the validated controller model, now assumed to be correct, will have a different execution when implemented on the control platform. This defeats the purpose of validation.

How to generate code from models such that the execution of these models on a real-time platform is similar to the validated behavior?

In summary, the four identified challenges which are addressed in this thesis are:

(1) How to exploit MBD and early integration for supervisory controllers (for BHSs) to achieve models with an increased quality and flexibility and a reduced complexity?

(2) How to provide user-feedback on the removal of states during synthesis in order to increase the usability of supervisory control synthesis in industry?

(3) How to perform model-to-model transformations, while retaining full traceability information on the requirements?

(4) How to generate code from models such that the execution of these models on a real-time platform is similar to the validated behavior?

1.8 Layout

The chapters of this thesis are directly related to the challenges introduced. First, Chapter 2 focusses on the modeling of supervisory controllers for BHSs. This chapter is an adapted version of Swartjes et al. (2017b). This chapter provides guidelines that result in an increased quality and flexibility and a reduced complexity of, especially, BHS models. When possible, the guidelines are generalized.

Second, Chapter 3 focuses on the generation of causes for the absence of states from the supervised system. This chapter will provide a formal system for the derivation of these causes and an algorithm that creates these causes during synthesis.

Third, Chapter 4 and Chapter 5 are dedicated to the relation between models which undergo model-to-model transformations. Where Chapter 4 defines the formal relationships, Chapter 5 uses these definitions and provides an alternative algorithm for the removal of synchronization.

Fourth, Chapter 6 shows the work done on the implementation of CIF models on real-time platforms, in which a specific platform, i.e., PLC, is leading. First a generic approach is put forward that is applicable to a significant subset of CIF models. The specific implementation made for BHSs is discussed last, which is adapted from Swartjes et al. (2017b). Creating successful real-time implementations was very important as it shows the industry, in this case Vanderlande, that the approach of this thesis can be applied.

Last, Chapter 7 provides the conclusions of this thesis.

Chapter 2

Modeling baggage handling systems

This chapter is dedicated to the creation of correct models. First, the modeling framework is introduced and the current state of the art techniques are investigated. Secondly, modeling consideration are defined that aid the modeler in the creation of correct models. Finally, ways of validating the correctness of created models are shown.

2.1 Framework

In this section, the creation of models by means of CIF is highlighted. To this end, a general introduction is given to the syntax and semantics of the hybrid automata used as the modeling entities in CIF.

2.1.1 Automata

Automata are (finite) state machines, that can be used to model software and hardware (Cassandras and Lafortune, 2009). An example of an automaton is depicted in Figure 2.1. This automaton models the detection of a blocked product by means of a photo-electric cell (PEC). Based on the PEC signal, it detects when a product is underneath the sensor and when a product gets stuck (an error). A stuck product is a product that takes longer to pass the PEC than a set time.

Syntax

In the setting of this project, an automaton consists of locations, events, edges, variables, differential and algebraic equations, and initialization predicates.

Locations model the different modes of the system. In Figure 2.1, three location are present: “Absent”, “Present”, and “Error”. These locations model, respectively, the absence or presence of a product under the sensor, or the presence of an erroneous (stuck) product.

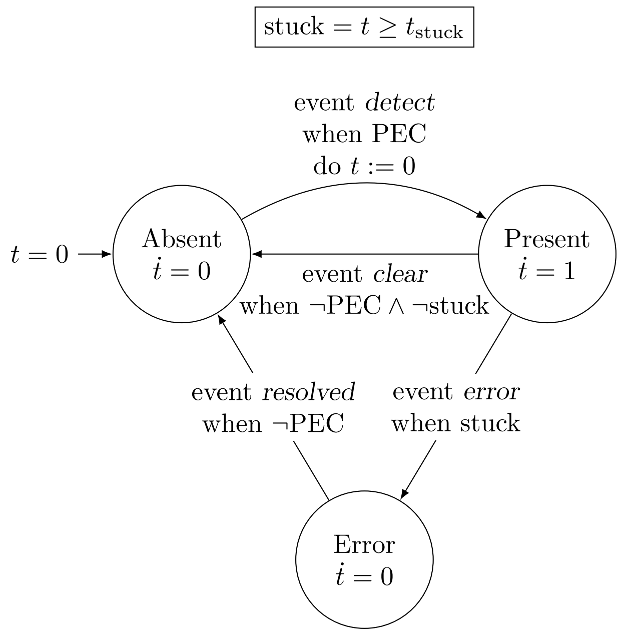


Figure 2.1: Automaton modeling the blockage detection of a product.

Events model the actions of the system. The automaton of Figure 2.1 has four events: *detect*, *clear*, *error*, and *resolved*. These events model, respectively, the action of detecting a product at the PEC, the action that a product is no longer detected by the PEC, a stuck product (error), and the action to notify that the error has been resolved.

Variables model the data of the system. There are three types of variables present in the framework: discrete, continuous, and algebraic variables. The value of a discrete variable only changes during a transition, as modeled by the update. The value of a continuous variable may also change during a transition, as modeled by the update. Additionally, the value of a continuous variable may change due to the passing of time. How the value of a continuous variable evolves, is defined by a differential equation. Finally, the value of an algebraic variable is defined by an equation. It will, at all times, reflect the value defined by the equation, and cannot be assigned a value. The equations defining the algebraic variables, i.e., the algebraic equations, are denoted in a box above the automaton as done in Figure 2.1.

The automaton of Figure 2.1 consists, among others, of three variables: t, PEC, and stuck. The continuous variable t models how long a product is underneath the sensor. Input variable PEC refers to the value of the sensor. If PEC is true (or high) there is a product beneath the sensor. If no product is present, PEC is false (or low). Algebraic variable stuck is true when a product resides too long beneath the PEC and is considered stuck.

Two additional remarks are made. First, note that variable PEC is an input variable, which means that the value of this variable is provided by an external source.

As a result, the variable can only be read and cannot be written. Second, note that t_{stuck} is not a variable but a constant whose value is assumed to be known.

As stated, time-related changes of a continuous variable are defined by one or more differential equations. In this framework, the differential equation may change per location. For instance, the derivative of t in location "Present" is one, while the derivative is zero in location "Error". To be complete, a differential equation must be defined for each continuous variable in each location.

Edges model the transitions of the system and are depicted as arrows. The arrow starts at the source location and points to the target location. Edges are labeled with a single event to denote the action that takes place during the transition. The event label is defined after the keyword "event".

To specify when a transition may occur, each edge contains a guard. The guard is defined after the keyword "when". The change of variable values is modeled by the update. The update is defined after the keyword "do". Updates are given as assignments, e.g., $x := y$.

A transition can only occur from an active location. The initial active location is denoted by a small incoming arrow. This arrow will also contain the initial values of the variables which can be written, i.e., algebraic and input variables will not appear here. When a transition occurs, the target location of the transition, i.e., the location to which the edge points, becomes active.

The CIF syntax of the automaton depicted in Figure 2.1 is as follows:

Listing 2.1: CIF syntax modeling the automaton of Figure 2.1

```
const real t_stuck = 1;

automaton detector:
    cont t = 0;
    alg bool stuck = t >= t_stuck;
    event detect, clear, error, resolved;

    location Absent:
        initial;
        equation t' = 0;
        edge detect when PEC do t:= 0
                    goto Present;
    location Present:
        equation t' = 1;
        edge clear when not PEC and not stuck
                   goto Absent;
        edge error when                 stuck
                   goto Error;
    location Error:
        equation t' = 0;
        edge resolved when not PEC goto Absent;
end
```

The syntax is self-explanatory. For the sake of completeness, `const` is used to denote a constant, `cont` is used to denote a continuous variable, and `alg` is used to denote an algebraic variable. Finally, `t'` is used to denote the derivative of `t` with respect to the time.

Semantics

In this part, the semantics of an automaton is given intuitively. To this end, the model of Figure 2.1 is considered. The semantics defines how the state of the model changes, due to the execution of transitions. The state of the system is defined by the active location and the values of the variables (valuation).

In this framework, all transitions are considered urgent. This means that no time may pass as long as at least one transition is possible. In essence, the execution of transitions is non-delayable. If multiple transitions are possible simultaneously, one transition is chosen non-deterministically.

To describe the semantics of Figure 2.1, an active state must be determined which is the initial location together with the initial values of the variables. In this particular case, two initial states are possible which only differ in the value of PEC:

(1) Location "Absent", a value of zero for t, and there is currently a product detected, implying a value of true for PEC.

(2) Location "Absent", a value of zero for t, and there is currently no product detected, implying a value of false for PEC.

For the transition labeled *detect* to occur, the guard must be satisfied. Either, the systems stays in location "Absent" until a product is detected, i.e., PEC becomes true, or event *detect* is directly triggered in the case that PEC is initially true. Executing *detect*, the value of t is set to zero and location "Present" becomes active.

When the location "Present" is active, the value of t increases by one per time unit as defined by the differential equation. Depending on the value of t_{stuck} and the moment the value of PEC becomes false, either the transition labeled *clear* occurs or the transition labeled *error* occurs. Note that, due to urgency, *error* is executed directly at the moment t becomes greater or equal to t_{stuck}.

If *clear* is executed, location "Absent" becomes active with t between zero and t_{stuck}. If *error* is executed, location "Error" becomes active with a t at a value of $t_{\text{stuck}} + \delta$, with δ a infinitely small (arbitrary) value. Note that t cannot have any other value, due to urgency.

2.1.2 Compositions

To create larger systems of multiple automata, composition is introduced. In a composition, multiple automata are executed in parallel. To allow interaction between automata, synchronization is used. Synchronization is related to the events of the system; equally labeled transitions must be executed simultaneously in all comprising automata, or the execution of the event is blocked.

Syntax

To show the syntax of compositions, an extra component is added to the system. This component models the following requirement: "After two errors, the next error shall sound the alarm until the system is reset." The composition of the latter requirements and the requirement of Figure 2.1 is depicted in Figure 2.2.

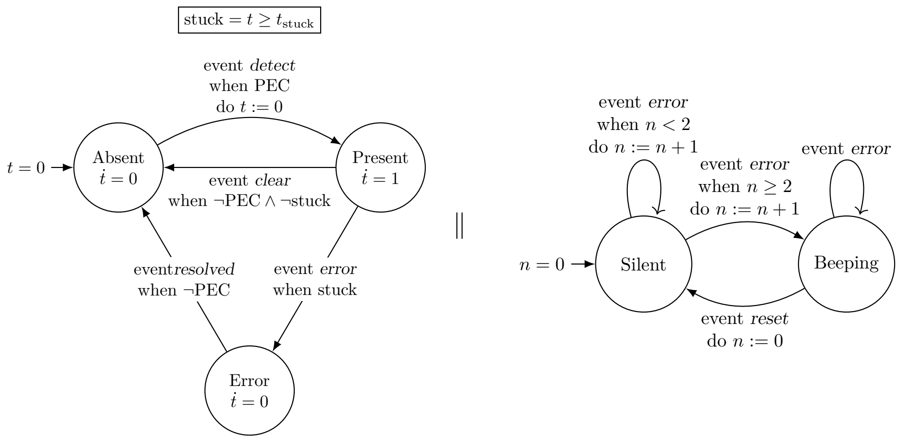


Figure 2.2: Composition of two automata.

In Figure 2.2, the parallel composition operator (||) is used to denote that the automaton left of the operator runs in parallel with the the automaton right of the operator, synchronizing on the shared events.

Semantics

The semantics of the composition in Figure 2.2, is given intuitively in this section. First, observe that the event *error* is present in both the left and right automaton. Such events are called shared events and must synchronize in a composition; only when in both automata the event *error* is possible, the associated transition may be executed.

Secondly, observe that events *detect*, *clear*, and *resolved* of the leftmost automaton and event *reset* of the rightmost automaton are not shared. Hence, the execution of these events is not influenced by the other automaton; the execution follows the semantics of an individual automaton.

In conclusion, when in location “Present”, the transition labeled *error* in the leftmost automaton cannot be executed until “stuck” is satisfied. Therefore, the execution of the equally labeled transition in the rightmost automaton is also blocked until stuck is satisfied.

At the moment “stuck” is satisfied, the transition labeled *error* may be executed. If executed, both automata change state simultaneously. The leftmost automaton makes a transition from “Present” to “Error”, while the rightmost automaton makes a transition from “Silent” to “Silent” if the number of errors if smaller than two. Otherwise, a transition is made from “Silent” to “Beeping”. The location “Beeping” remains active as long as event *reset* has not occurred.

2.2 Related work

2.2.1 General overview

The research field on MBD is very wide. Related terms are Model-Based Systems Engineering (MBSE), Model-Based Software Engineering (MBSE), Model Driven Development (MDD) and Model Driven Engineering (MDE). In software design, the term Model Driven Architecture (MDA) is used by the Object Management Group (OMG), that has defined the well-known Unified Modeling Language (UML) and System Modeling Language (SysML). The interested reader is referred to Russo (2016) for a discussion of and references to these various terms.

A well-kown tool for MBD is Simulink by MathWorks for continuous-time modeling, and Stateflow for discrete-event modeling based on a specific version of Statecharts. Statecharts were originally developed by Harel (1987), and are now available in many different forms and dialects, see for example von der Beeck (1994). Stateflow has no formal semantics, but the graphical toolset is user friendly.

A tool that does have a formal semantics, and is also used in critical industrial applications is the SCADE Suite (Esterel Technologies, 2017; Camus, 2013). It uses data flow modeling for supervisory controller design. Plant models, however, cannot be modeled in this way. For this purpose, a separate tool Simplorer is available, and Simplorer models can be connected to SCADE Suit models via cosimulation.

The cosimulation approach is also available in MATLAB Simulink via the S-function interface. Even the integration of Stateflow and Simulink is based on the S-function interface. Note that integration by means of the S-function interface implies a master-slave relation, where Simulink is the simulation master, which controls the simulation, and the model that is encapsulated in the S-function, acts as the slave.

Cosimulation aims at providing interoperability of a wide range of different simulation models and tools. It differs from our design framework, which is based on a *single* toolset and modeling language that are suited to both control system and plant modeling, simulation, visualization and code generation.

Whittle et al. (2014) discuss the state of practice in model driven engineering. They note that, perhaps surprisingly, the majority of MDE examples in their study followed domain-specific modeling paradigms. Their interview data showed that it is common to develop small domain-specific languages (DSLs) for narrow, well-understood domains. They also notice widespread use of mini-DSLs, even within a single project, and subsequently observe a clear challenge how to integrate such multitude of DSLs. The use of such DSLs contrasts with our approach, which is based on a more general purpose modeling framework for supervisory control system development.

Another overview study is from Vyatkin (2013), who presents a 'State-of-the-Art Review' of 'Software Engineering in Industrial Automation'. This study focuses on control system design only. It does not discuss methods or tools to use plant models for model-based testing, to shorten the control system development process, as proposed by our framework. The article does, however, discuss several relevant standards and norms for the development of industrial automation software, including the IEC 61131-3 PLC standard that is used as one of the backends of our framework for code-generation.

2.2.2 Baggage and material handling

There is also related work on specifically the (re)design of BHSs and material handling systems. Cavada et al. (2017) have created an integrated simulation model of the international airport terminal of Santiago, Chile. The goal of this simulation model is to investigate how the current system can be improved and is dedicated to investigate (possible) flow and capacity problems. To this end, a vehicle traffic simulation model is extended. Determining capacity problems and investigating the flow of a BHS is an important part, and is crucial to determine the properness of the controller.

The approach given in Cavada et al. (2017) focuses on the validation of, so-called, liveness requirements. As our approach is concerned with the complete design of the controller, compliance to requirements related to safety issues must also be investigated. An example is the injection of foreign objects in the system, other than from the check-in counters. Moreover, the actual implementation of the resulting controller is not considered, whereas we are also concerned with the creation of code for the control platform.

Johnstone et al. (2015) focuses on simulation and validation of different merge strategies. This is done, more in the nature of our approach, on low-level control where sensor signals are used to determine the appropriate control actions. Although low-level control is considered, nothing is stated on how the result can be implemented afterwards. Also, their approach is limited to a single part of a BHS, whereas in our approach the complete BHS is considered. Nevertheless, the approaches provided on the reconstruction of unobservable states, i.e., positioning of products on the belt, are closely related to the considerations that are provided in this thesis.

Using the IEC 61499 standard, Black and Vyatkin (2010); Yan and Vyatkin (2011) use a similar approach to the creation of a BHS controller as used in this thesis. For the BHS, a separation is made between the part that models the uncontrolled system, the controller, and a part that can reconstruct the system state for validation purposes. Because the IEC 61499 standard is executable, validation has been used to assess the quality. A major difference between Black and Vyatkin (2010); Yan and Vyatkin (2011) and the approach of this thesis, is the fact we strives for an end-platform independent model. Our view is that the model should be independent of specifics associated with the control platform until the actual controller is built (or generated). Otherwise, the model may be influenced by the properties of the control platform deteriorating the quality and flexibility. Note that of course some properties of the control platform must be considered. However, in our view, it should not matter in the end whether a PLC with an older standard or an Industrial PC with a C implementation is used.

A different direction is shown in Rijsenbrij and Ottjes (2007) in which model-based design and simulation is used to aid the development of new concepts for BHSs. In this paper, a model is made for a vehicle that moves baggage from a carrousel to an aeroplane. This model has been used to determine how much could be gained from using this concept. This model is not (yet) concerned with the implementation phase, and hence is built for a different purpose than the models that are shown in this thesis. Nevertheless, the importance of being able to assess the quality before the system is built is shown.

Both McGregor (2002) and Johnstone et al. (2007) show how emulation techniques can be applied to BHSs. Whereas Johnstone et al. (2007) focuses more on the ability of allowing such large-scale emulations, McGregor (2002) also discusses the gained benefits of using such techniques. It is of importance that emulation can be executed on such large systems. We have also used emulation to show correctness of our models, as will be shown later. However, before emulation can even commence, first the controller has to be built. Even with sophisticated emulation models, it remains difficult to pinpoint errors and adapt the controller accordingly when errors are found. Preferably, the quality is assessed earlier in the design approach.

2.2.3 Proposed model-based framework

Much of the previous work is related to the development of the CIF formalism and its associated toolset. One of the major achievements of CIF, is that its formal design has allowed the implementation of several algorithms for synthesis of supervisory controllers, based on formal requirements and formal plant models, in such a way that the resulting synthesized supervisors are nonblocking and satisfy the formal requirements by construction.

This supervisory control systems framework has been applied in several industrial cases. We mention a number of them below.

- In (Theunissen et al., 2014), supervisory control synthesis is applied to a support system used to position patients in an MRI scanner of Philips Healthcare (`http://www.medical.philips.com`).

- In the Océ printer case study, reported in Markovski et al. (2010a), supervisory control synthesis is applied to the scheduling of maintenance operations in a high-end printer of Océ Technologies (`http://www.oce.com`).

- In the theme park vehicle case study, reported in Forschelen et al. (2012), supervisory control synthesis is applied to the multimovers of ETF (`http://www.etf.nl`).

- Recently, new projects for the design and supervisory control of waterway locks have been started in cooperation with Rijkswaterstaat (`https://www.rijkswaterstaat.nl/english`), which is responsible for the design, construction, management and maintenance of the main infrastructure facilities in the Netherlands. First results on supervisory control synthesis are reported in Reijnen et al. (2017).

A theoretical contribution is the work of Swartjes et al. (2014), that investigates the creation of controller code from automata models comprising synchronization. Although the concept and need for synchronization have not been discussed yet, it proves difficult to create controller code from models with synchronization without changing the structure of the model and therewith the traceability of requirements.

In this work, model-to-model transformations are proposed that better retain the original structure of the model.

Please note that MBD for material handling systems can also be applied in a somewhat different form than is presented in this thesis. This has been done by Swartjes et al. (2017a) for a high-volume printer where the complete system is described by a set of mathematical formulas describing the dynamics of sheets within a printer. By allowing design variables related to the control strategy, e.g., the accelerations of motors controlling the movement of sheets, as well as the physical design, e.g., the length of the paper track within the printer, both the control strategy as well as the printer dimensions could be determined in an optimal fashion by using standard optimization techniques.

2.3 Considerations

In general, the same system may be described by different models. This also holds true for BHSs. However, some modeling choices may lead to an increasing complexity, and/or a decreasing quality and flexibility. Therefore, modeling considerations are presented in this section for the modeling of BHS systems, based on our experiences. The goal of these consideration is to reduce complexity, increase quality, and increase flexibility of resulting models. Although these considerations are devised specifically for BHS systems, a generalized form, if possible, is also provided.

2.3.1 Quality

The following considerations are all related to the quality of the model.

Synchronization

In a previous section, the concept of synchronization was introduced. Although only a small example was provided, the takeaway message is that synchronization is a way of creating a (more) modular model: using synchronization, it is not necessary to directly model the composed behavior or dynamics.

Especially for BHSs, allowing a modular approach for the implementation of requirements influences the quality of controller. Namely, even a simple section, like a TRS, may contain over twenty different requirements. Without a modular approach, the quality of the implementation may not be assessed: it may prove very difficult to determine from a model of the combined requirements whether or not all requirements have been implemented. As seen with the example of Denver International airport, not being able to determine the quality of the model may result in disastrous results. The same has been observed by Grigorov et al. (2011) and Haoues et al. (2016).

A situation in which synchronization really aids the modeler is when requirements must be implemented on both the level of section and zone. For instance, consider the following requirements:

1. A section must be stopped when a stop signal is given.

2. A zone must be stopped when a stop signal is given.

3. When a zone is stopped, all associated sections must be stopped simultaneously.

The difficulty lies in the fact that a section must follow the behavior of the zone immediately. In a framework which does not support synchronization the conditions must be duplicated, as can be seen in Figure 2.3.

In Figure 2.3, the left-hand side models the zone which must stop when the zone signal is high: it must reach the location Stopped. The right-hand size models the section which must stop, i.e., reach the location Stopped, when either the section or the zone signal is high. Note that without synchronization, requirements 2 and 3 are implemented together. Although in the example this duplication can be easily performed and the quality can still easily be assessed, remember that a zone normally does not consists of a single section. Hence, if the condition on stopping the zone changes, all sections must be manually changed, which may induce errors in the model. Moreover, it may become more difficult to determine whether all requirements are implemented.

A more desirably situation in our perspective is obtained when synchronization is used as can be seen in Figure 2.4.

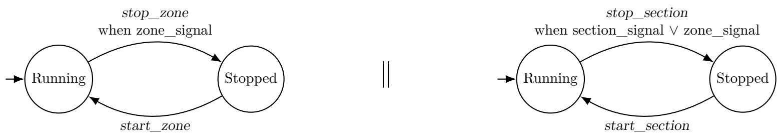


Figure 2.3: Duplication of guards when synchronization is not allowed.

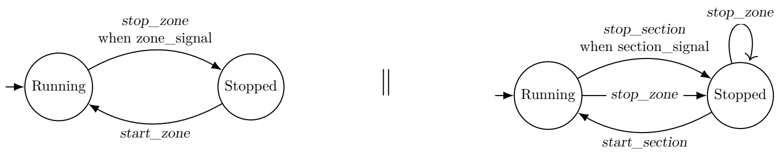


Figure 2.4: More desired and traceable implementation of requirements.

In Figure 2.4, again the left-hand side represents the zone while the right-hand side represents the section. When synchronization is used, solely the event *stop_zone* can be used to model a transition to the location Stopped. The actual condition when *stop_zone* occurs is all modeled in the zone automaton. To prevent blockage, as will be discussed in the following section, the selfloop is added to the Stopped location to allow synchronization with the event when some sections are already stopped.

Synchronization allows for a per-requirement approach, where each requirement may be modeled as a separate automaton synchronizing on shared events. In such a way, a completely clear and unambiguous relation between requirements and automata is created. In other words, a complete one-on-one traceability relation can be obtained. With respect to the definition of quality, it is very straightforward to determine whether or not all requirement are implemented in a correct manner.

To increase quality, i.e., being able to assess the software quality, use synchronization to allow for an unambiguous traceability relation between the model and requirements to cope with the numerous requirements a system comprises.

Although synchronization can be used to increase quality, synchronization may also have a negative influence on the quality (and complexity of the model) as will be discussed in the next section. Nevertheless, the consideration, as stated, remains valid.

Blockage

Although synchronization is a powerful concept, it also may become a major source of errors when many automata synchronize on the same event: when multiple automata share the same event, it becomes more difficult to oversee all situations in which undesired blockage may occur. As a result, the complexity of the model may even increase and the quality of the model may decrease.

To reduce the chance of an undesired blockage and, hence, prevent the reduction of quality, the number of constraints per synchronizing event must be minimized as much as possible. In such a way, the situations under which synchronization occur remain insightful and predictable. This can be achieved by allowing only one automaton to pose conditions on the execution of an event. In other words, for each event only one automaton may contain a guard for that event while all other automata may not block the event. As an example, consider the composition of Figure 2.2 in which the transitions associated with the events *error* are blocked in particular situations in the leftmost automaton. The rightmost automaton, however, does not block the associated transitions: in each location the transitions associated with event *error* is possible. As a result, the conditions under which synchronization occur are more clearly derivable from the model. As a benefit, the source of errors can be found quicker as the conditions on synchronization are concentrated on a single location. A similar approach was applied for the model of Figure 2.4.

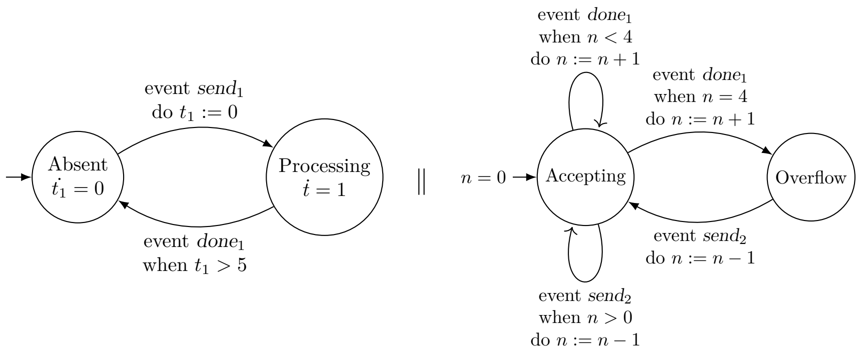


Figure 2.5: Buffering system.

Although this guideline reduces the change of undesired blockage and therefore increases quality, adhering to this guideline may be difficult and may even, in its turn, negatively influence quality. For example, it is difficult to comply to this guideline for buffering systems. Such a system is depicted in Figure 2.5. When an overflow occurs, the buffer cannot accept a product. Additionally, the condition under which a product can be sent to the buffer is also guarded. If the guideline is to be adhered to, modularity must be sacrificed by combining all conditions on a single edge. As a result, the guideline is stated in a less restrictive way:

> To prevent a reduction of quality, reduce the occurrence of event-specific guarded transitions to a minimum of different automata when possible.

Or, in other words, prevent unnecessary placement of event-specific guarded transitions in a multitude of different automata. All in all, a balanced choice must be made between the latter two guidelines to end up with a model for which quality can be straightforwardly assessed.

Coding errors

After a model is created and validated, at some point the model must be converted to code that can run on the control platform, i.e., the controller must be built. Due to the sheer size of BHSs and, hence, the sheer size of the associated models, manual encoding of a controller still allows for many errors. Trivially, coding errors can be prevented when an automatic approach, i.e., code generation, is chosen. As a result, a very straightforward and trivial guideline is stated:

> To increase quality, use code generation to automatically create the resulting controller code to prevent coding errors.

2.3.2 Flexibility

The following considerations are all related to the flexibility of the model.

Observers

Observers are automata that, based on information available, reconstruct derivable information. Normally, requirements can only be implemented when the associated states are directly observable. For a conveyor belt this is, in the most basic case, solely the current state of the PECs. Based on these signals, all requirements must be implemented. Hence, as already mentioned earlier in this thesis, a requirement on the number of products on a belt or the exact position of a baggage item cannot be straightforwardly implemented without additional measures. As such, observers allow for flexibility in the model in the broad sense of the definition.

In control theory, observers are a well-known solution for the reconstruction of unobservable states (Luenberger, 1971). To allow for implementation of temporal

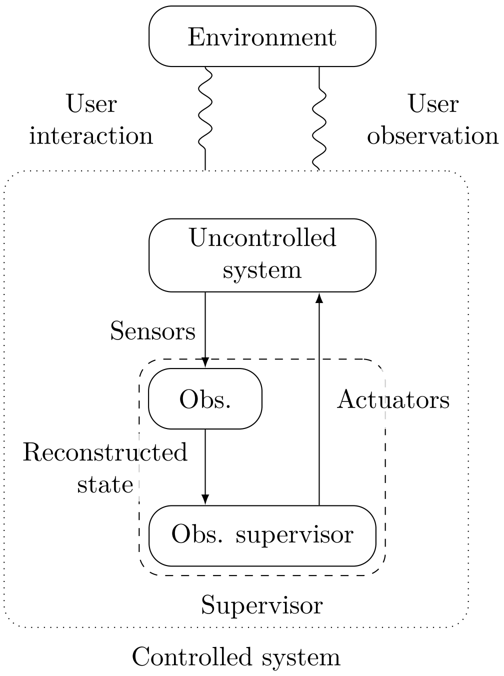


Figure 2.6: Schematic overview of a controlled system, incorporating an observer.

requirements, based on solely the discrete information of PECs, observers will be used resulting in a control scheme as shown in Figure 2.6.

Using observers, the number of products on a transport section can be reconstructed by counting the number of received and sent products. An example is provided in Figure 2.8, where the event *send* is used to count the number of occurrences. Using this observer, the state of the system is extended with the value of m that represents the number of products in the system.

To increase flexibility, observers can be used to reconstruct the unobservable states of BHSs to provide a straightforward implementation of all requirements.

Requirements as features

Considering a complete BHS, the set of requirements may change slightly per section based on either the downstream or upstream section or the encapsulating zone. As an example, a transport section that is part of a merge zone may need to stop a product just before the end to create necessary gaps while a "standard" transport section does not need this functionality. Additionally, requirements of a section may change due to user-specific requirements based on legislation.

It is desired to allow a changeable set of active requirements per transport section.

Otherwise, a separate transport section must be modeled for every combination of active requirements. To allow such a changeable set of requirements, a feature-based approach is desired which is possible also with help of the concept of synchronization. The idea is taken from software engineering, where features are used denote items which increase functionality for a specific system (Benavides et al., 2010; ter Beek et al., 2016).

Features allow for straightforwardly change or add behavior of, for instance, transport sections. An example is the addition of requirements for transports sections when part of a specific zone. These additional requirements may be considered features that can be either present or not, based on the encapsulating zone. As a result, the flexibility of the design increases using a feature-based implementation.

The idea of modularization is schematically depicted in Figure 2.7 in which one particular section is denoted. The interface is defined by two events, i.e., $send_{k-1}$ and $send_k$. This is how the section interacts with the other sections and/or zones. The key idea is that each feature can be added or removed as desired within the specified interface. The chosen features will form a composition in the model and can only synchronize with the specified interface events.

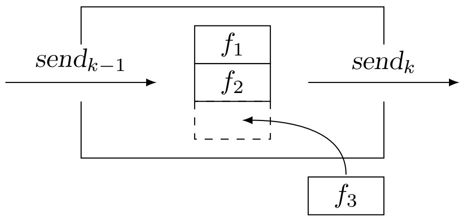


Figure 2.7: Schematic overview of features in a module.

Using the specified interface of Figure 2.7, possible features are depicted in Figure 2.8, Figure 2.9, and Figure 2.10. Note that, following the guidelines, only $send_{k-1}$ in Figure 2.10 is guarded and may potentially block the transition. Moreover, note that these features are not completely independent; the feature of Figure 2.8 is needed for both Figure 2.9 and Figure 2.10 as they both depend on the number of products on the belt (m).

To increase flexibility, use synchronization to allow a feature-based implementation of requirements to cope with the need of adaptability of BHSs.

For completeness sake, the feature of Figure 2.9 is explained in more detail. Figure 2.9 models an energy save feature: “If no product is present and no new product has arrived within four seconds, energy save must be turned on.” There are two Off locations, both modeling the fact that energy save is Off: Off_1 models the fact that there are no products on the belt, while Off_2 models the fact that there are products on the belt. Switching between the two Off states happens, based on the events and the observer depicted in Figure 2.8. When location Off_1 is active for four seconds, location On becomes active. This location is left as soon as a new product is received.

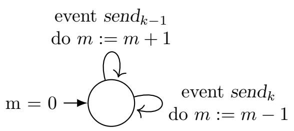


Figure 2.8: A counting observer; keeps track of the number of items.

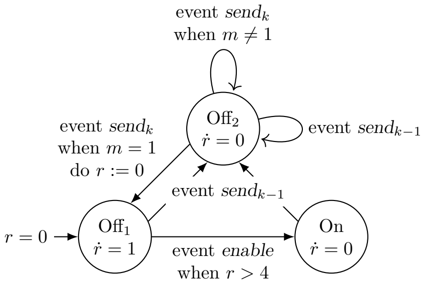


Figure 2.9: Energy save requirement; turn on energy save if no product is present and no new product has arrived in the last 4 seconds.

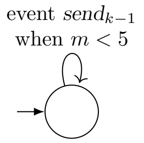


Figure 2.10: A requirement to prevent overflow; a maximum of 5 products is allowed.

Product- or equipment-based models

In a BHS, many products roam the system which, upon the arrival at PECs, discretely change the state of the system. As a result, it has been found that BHS models may be focused on either the products or the equipment. For the sake of clarification, each baggage item product and each transportation unit component is modeled separately in a product-based model. Based on its position in the system, the dynamics of the associated transportation unit component is adopted. Note that this implies that all products are global and that they must be aware of all components in the system with which they can potentially interact. Moreover, each product must be aware of every potential neighboring product if a feature like collision detection is to be implemented. Schematically, in Figure 2.11 an overview is given of the components, products and their interactions. An example of a model which uses a product-based approach is given by van der Sanden et al. (2015), in which the wafers of the lithography system are modeled separately.

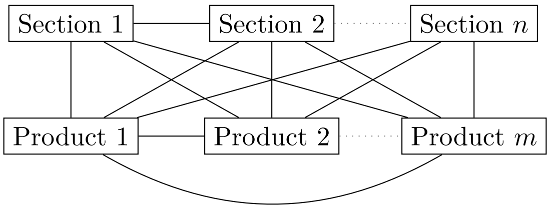


Figure 2.11: Product-based model.

In an equipment-based model, each component interacts only with its own internal products. Products enter the component via a hand-over from their predecessor component, and exit the component via a hand-over to their successor component. Often, when an equipment-based model can be adopted, the dynamics of the internal products can also be reduced to a one-dimensional representation. When no product can overtake another products, suddenly disappear, or overlap, a strict order can be derived for the products. This further reduces the complexity needed to model the hand-over of products between two components; the first product, for instance, is always the most downstream product. Schematically, in Figure 2.12 an overview is given of the components, products and their interactions. Note that the number of products per component may be parameterized.

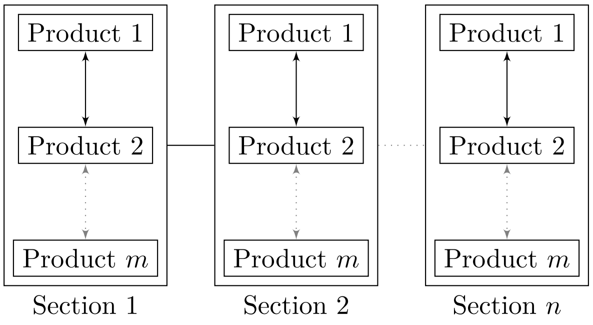
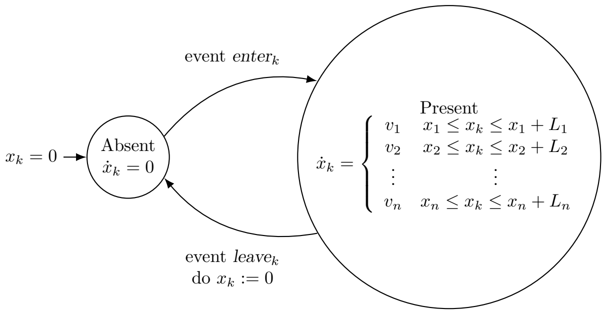


Figure 2.12: Equipment-based model.

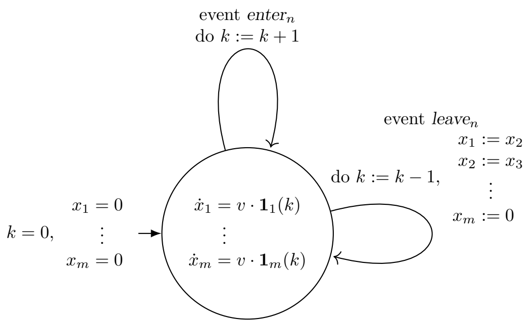
For the sake of simplicity, the differences between a product-based and equipment-based model are shown using two different hardware models, i.e., models that are used to simulate the plant. As this models the plant, the exact product positions of products are known (observable); in the plant model the state of the environment is completely observable while the state of the controller (except for the actuator signals) is not observable. The product-based implementation is given in Figure 2.13, while Figure 2.14 depicts the equipment-based implementation.


event $enter_k$
$x_k = 0$
Absent
$\dot{x}_k = 0$
Present
$\dot{x}_k = \begin{cases} v_1 & x_1 \leq x_k \leq x_1 + L_1 \\ v_2 & x_2 \leq x_k \leq x_2 + L_2 \\ \vdots & \vdots \\ v_n & x_n \leq x_k \leq x_n + L_n \end{cases}$
event $leave_k$
do $x_k := 0$

$$\begin{aligned} \mathrm{PEC}_1 &= \bigvee_{m \in M} x_m \geq x_{\mathrm{PEC}_1} \wedge (x_m - L_{\mathrm{prod}}) \leq x_{\mathrm{PEC}_1} \\ &\vdots \\ \mathrm{PEC}_n &= \bigvee_{m \in M} x_m \geq x_{\mathrm{PEC}_n} \wedge (x_m - L_{\mathrm{prod}}) \leq x_{\mathrm{PEC}_n} \end{aligned}$$

Figure 2.13: An automaton model that is product-based.

In Figure 2.13, an automaton is depicted that described a single product. In the



event $enter_n$
do $k := k + 1$
event $leave_n$
do $k := k - 1$, $x_1 := x_2$, $x_2 := x_3$, $\vdots$, $x_m := 0$
$\dot{x}_1 = v \cdot \mathbf{1}_1(k)$
$\vdots$
$\dot{x}_m = v \cdot \mathbf{1}_m(k)$
$k = 0$, $x_1 = 0$, $\vdots$, $x_m = 0$

$$\mathrm{PEC}_n = x_1 \geq x_{\mathrm{PEC}} \wedge (x_1 - L_{\mathrm{prod}}) \leq x_{\mathrm{PEC}}$$

Figure 2.14: An automaton model that is equipment-based.

actual model, m of these automata will be present in which k adopts the identifier of that specific instance. The variables x_1 through x_m are used to model the head of the product; x_1 through x_m model the position of the leading edge of a product. Note that the model must also contain n transport unit automata to model the, for instance, conveyor belts that transport these products.

A product can enter the system and afterward leave the system. While the product is present, the velocity of the projected component (v_1 through v_n) is adopted based on the current position of the product (x_k). The lengths of the sections (L_1 through L_n), are assumed to be known. The common algebraic equation for the PEC of each component is given in the lower part of the figure using L_{prod} as the (known) length of the product.

In Figure 2.14, the equipment-based variant is depicted that models a collection of products. Herein, $\mathbf{1}_n(x)$ is the Heaviside function which has a value of zero for all $x < n$ and a value of one for all $x \geq n$. It is chosen that x_1 models the product nearest the PEC. Hence, the algebraic equation modeling the PEC only needs to consider x_1. As a strict ordering is needed to reduce the number of interactions, the product positions must be shifted after a product leaves the component in order to retain this ordering.

The difference in the two methods resides in the difference in flexibility of the two models. In this case, flexibility is meant in the broadest sense of the definition: the ease of modifying the model to incorporate additional components and/or products. If the number of interactions in the models is compared, it is clear that the product-based approach has $M \cdot (M + N - 1) + N - 1$ interactions, whereas the equipment-based approach has $M \cdot N - 1$ interactions. Adding a component results in an addition of $M + 1$ interactions in the product-based approach, compared to a single interaction[1] in the equipment-based approach. Hence, the equipment-based approach has a higher flexibility; it is easier to add a new component. The same holds for a change in the

[1]The internal interactions between the products are not considered in this case as they are part of the automaton.

number of products, with a respective change of $2M + N$ interactions versus one.

The number of variables needed differs between the two approaches. Where the product-based approach only needs $M + N$ variables, the equipment-based approach needs $M \cdot N$ variables. Although the theoretical difference in the number of variables (and therefore also states) can become very large, the practical difference will be nil for many systems; normally, the maximum number of products is based on the maximum number of products the complete system can contain, i.e., the number of components times the number of product each component can contain. As a result, in many practical cases the number of variables needed to describe the system is equal for both approaches.

This guideline can be generalized to any system in which products can roam the system (freely) while being bound to component-specific dynamics like, for instance, a manufacturing line.

To increase flexibility, or the ease with which components can be added or removed from the system, adopt an equipment-based modeling approach for BHSs.

Or, in other words, the model can be more easily adapted when requirements change if equipment-based modeling is favored.

Control platform

As airports are located throughout the world, legislation in the different countries may put restrictions on the control platform that can be used for the implementation of the controller. However, it is undesired that the complete controller has to be redesigned for each of the control platforms in existence. Namely, errors may be (re)introduced due to conversions between platforms.

Code generation also provides a solution for this particular problem; based on a common model, code generation may provide the generation of code for different control platforms. This will result in controllers that behave the same while being deployed on different control platforms. Note that lots of effort must be put in once to create a generator for a specific back-end. However, subsequent usage does not need the same amount of effort.

To increase flexibility, use code generation that allows for the generation of code for different control platforms.

2.4 Validation

In this section, validation of a BHS in a model-based framework is discussed. To this end, in addition to the model of the controller, a model of the uncontrolled system or the hardware (also sometimes referred to as a plant) is added to allow simulation of the interaction between the uncontrolled system and the controller. Note that the solely the controller can be validated, but this will omit the dynamics of the plant which may be very important to assess the correctness.

During validation, additional observers can be added to the system that only aid the validation phase. Finally, different visualization techniques are shown to assess the quality of the system. Remember that, for the sake of completeness, assessing the quality of the system in the traditional design approach could only be done after the controller was coded.

2.4.1 Hardware models

For validation purposes, a model of the hardware is coupled to a model of the software. To allow for a modular design, an interface is defined as specified in the guidelines. For the sake of implementation purposes, it is desired to model the interface based on the actual interface present between the hardware and software.

Considering a TRS (Figure 1.2), the interface is defined by the signals PEC and MTR. The software defines when the motor should start or stop, so MTR is an output of the software and an input to the hardware. For the photo-electric cell (PEC), the converse is true.

$$\boxed{\begin{array}{rcl} y & = & x - L_{\text{prod}} \\ \text{PEC} & = & x \geq x_{\text{PEC}} \wedge y \leq x_{\text{PEC}} \end{array}}$$

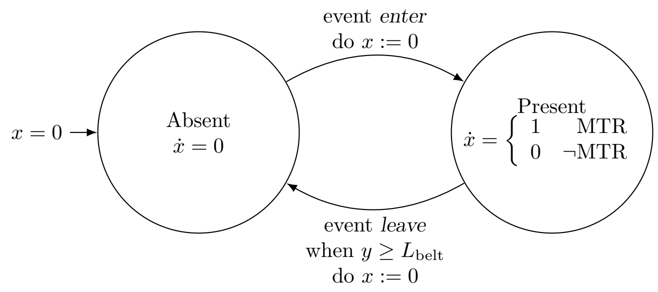

event *enter*
do $x := 0$
$x = 0$
Absent
$\dot{x} = 0$
Present
$\dot{x} = \begin{cases} 1 & \text{MTR} \\ 0 & \neg\text{MTR} \end{cases}$
event *leave*
when $y \geq L_{\text{belt}}$
do $x := 0$

Figure 2.15: Automaton model of the hardware.

A possible model describing the hardware of a TRS is depicted in Figure 2.15. In this model, the signal for the PEC is depending on the input signal MTR via variable x. Note that the position of the PEC, i.e., x_{PEC}, the length of a product, i.e., L_{prod}, and the length of the conveyor belt, i.e., L_{belt}, are assumed to be known. Note that this model resembles Figure 2.14 considering only one product.

The two locations model the presence or absence of a product. The continuous variable x models the position of the front of the product and the algebraic variable y models the position of the back of the product. The position x (and therefore also y) increases if a product is present and the motor is on. The PEC signal is defined as an algebraic variable: if the product is beneath the photo-electric cell, the value of PEC is **true**. Finally, when the back of a product reaches the end of the belt, the product leaves the belt.

2.4.2 Observers for validation

In addition to hardware models, observers are used to reconstruct the unobservable parts of the system. In the broad sense of the term “unobservable state”, two additional types of observers are provided in the following two sections, namely, observers related to safety and observers related liveness requirements. For safety, it can be observed whether a certain violation has occurred. For liveness, it can be observed whether a certain desired goal has been reached, e.g., throughput. In short, these specific kinds of observers will increase the ease of assessing the quality of the system.

Safety

As stated, an observer related to safety must be able to determine whether something bad has occurred. In essence, this type can be seen as an assertion; when the condition is violated, a transition occurs to denote the violation. Hence, a very simple example is given in Figure 2.16. Whenever an error occurs, assuming error is an event that is synchronized with other error events in the model, the state changes from Safe to Unsafe. By determining the state of the automaton, the violation of the requirement can be assessed. When a simulation environment is used that allows for dynamic starting and stopping of simulations, the change of state can be used to stop the simulation. Together with a log file, the reason of violation can be analyzed and determined.

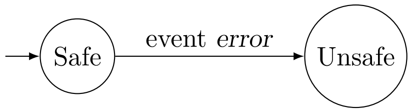


Figure 2.16: Example of an observer for a safety requirement.

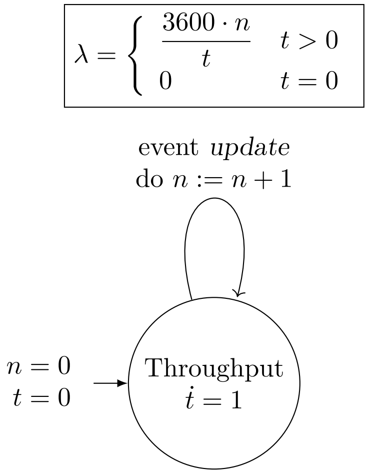
Liveness

Liveness requirements, like the throughput of the system, can also be validated in the current approach. An example is provided by the automaton depicted in Figure 2.17.

The observer is in fact a clock and a counter that increases each time a product enters a certain part of the system. The algebraic variable λ, with associated equation, determines the envisioned throughput in products per hour. Note that the event *update* must be substituted with the event for which the throughput must be determined.

2.4.3 Simulation

For the sake of simplicity, solely the composition of the controller from Figure 2.2 with the hardware model of Figure 2.15 is considered for the remainder of this section. It is assumed that $t_{\text{stuck}} := 1$, $x_{\text{PEC}} := 4$, $L_{\text{prod}} := 1.5$, and $L_{\text{belt}} := 6$. Additionally, the algebraic equation defining the motor signal is given as MTR $= \neg$Error.



$\lambda = \begin{cases} \frac{3600 \cdot n}{t} & t > 0 \\ 0 & t = 0 \end{cases}$
event *update*
do $n := n + 1$
$n = 0$
$t = 0$
Throughput
$\dot{t} = 1$

Figure 2.17: Example of an observer for a liveness requirement.

There are two possible ways of visualizing the state of the simulation. Either by plotting the values of the variables in a graph, or by graphically representing the system and adapting elements of the drawing to reflect the state.

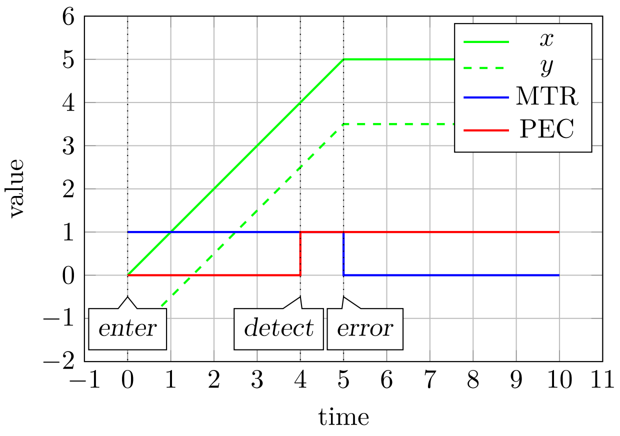


Figure 2.18: Output of the simulation depicted as a graph.

An example of such a graph, based on a real simulation, is depicted in Figure 2.18. Note that the value of x and of y, i.e., the front and the back of the product, are non-observable and only shown for the sake of completeness; the software can only observe the value of the PEC, provided by the hardware via the interface. A similar plot can be made of a safety observer, showing whether or not a requirement has been violated with respect to the simulation time.

As stated, the length of the product is 1.5. This can be observed in Figure 2.18, as the y value, at any time instance, is 1.5 units less than the x value. At time instance

0, the motor is running, which can be observed by the fact that MTR has the value one. Since the motor is running, the value of x (and y) increases. At time instance 4, the condition $x \geq x_{\mathrm{PEC}}$ is satisfied and the PEC is triggered. This can be observed by a rise in the value of the variable associated with the PEC.

From the software model, i.e., Figure 2.1, it is known that while the PEC is on, a timer is active. If the PEC does not go off within the time frame specified, i.e., t_{stuck}, an error is triggered. In this simulation, due to the length of the product, indeed an error occurs. Because of the error, the motor is switched off and the product stops moving.

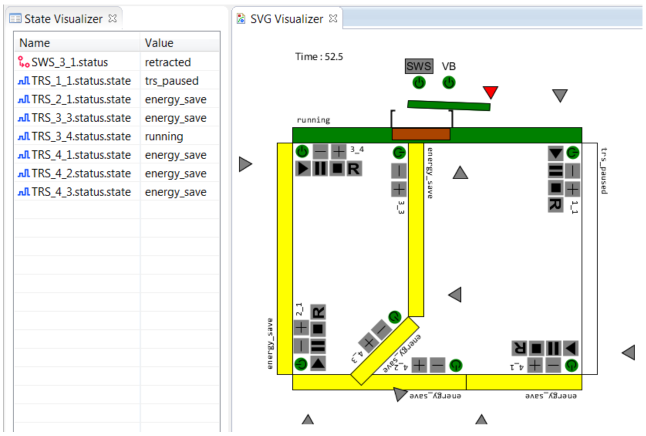
In Figure 2.18, the presence of an error is easily detected. The time underneath the sensor crosses the boundary of one second, which triggers an error. However, how to proceed is challenging without further knowledge of the system:

(1) Is this desired behavior, i.e., the product was too large for the system?

(2) Is the value of t_{stuck} wrong, i.e., too tight?

(3) It the requirement wrong, e.g., should the value of t_{stuck} depend on the (expected) length?

For reason (1), no solution exists as the user did something wrong. t_{stuck} was clearly designed to allow only products of one meter, which the user violated. For the reasons (2) and (3), early integration indeed assesses the quality of the system in an early stage of the design. This reduces the resulting design effort of fixing the errors, since the model can still be swiftly adapted (as it is still not too large) and the influences of the associated changes can be quickly investigated (a solution for both (2) and (3) can be made and compared).



Name	Value
SWS_3_1.status	retracted
TRS_1_1.status.state	trs_paused
TRS_2_1.status.state	energy_save
TRS_3_3.status.state	energy_save
TRS_3_4.status.state	running
TRS_4_1.status.state	energy_save
TRS_4_2.status.state	energy_save
TRS_4_3.status.state	energy_save

Figure 2.19: Output of the simulation with a graphical representation.

For larger systems, a graphical representation of the system is more suitable. An example of a such a visualization is depicted in Figure 2.19. Herein, a small circular BHS is depicted that has an inner and an outer loop. The visualization is based on Scalable Vector Graphics (SVG) that consist of a graphical representation of the system and the system components. These representations can be adapted to reflect the current state. For instance, the position, the scale, or the color of each conveyor belt can be changed accordingly.

In Figure 2.19, one product is currently present, as can be seen in the topmost TRS. This product is moving, as the green color of the associated TRS denotes a running state. Note that the state is also textually present in the left of the TRS. The panels near the TRSs can be used to change the operation mode of each TRS individually. It is possible to activate, deactivate, resume, pause, or stop the system. These actions are directly related to the HMI interface. Additionally, for simulation purposes, the addition and removal of products is possible with the, respectively, minus and plus buttons.

In this particular simulation, the energy save function of the system is investigated. As the product is currently on a long conveyor belt, it is expected that all TRSs enter the energy save mode. Moreover, as the product follows the outer loop, it is expected that the TRSs of the inner loop remain in energy save mode. Note that the rightmost TRS is manually put in a pause mode, and must remain in this mode until the user restarts the TRS.

The correctness of the expected behavior can be easily observed by only looking at the colors. This shows the effectiveness of using visualization in assessing the quality of the system.

2.4.4 Dependencies

Another validation method is provided by graphically representing the dependencies between components. Such a dependency can be due to, among other things, shared events or shared variables. Such a graphical representation of the dependencies is called an iconic overview.

Based on the need of a hierarchical decomposition, an implementation was made for CIF based on GraphML (Brandes et al., 2001). GraphML allows for hierarchical graphs from which, on demand, components and entities can be hidden; the level of detail can be changed upon request. The GraphML file can be read by an external application, like yEd from yWorks, to visualize the actual graph.

Iconic overviews are handy to quickly spot inconsistencies in the architecture of the system. As an example, one can observe whether a TRS is connected properly to its predecessor and successor. This will be shown by means of the following use case.

In systems with a lot of similar components, like a BHS, it is common that new components are introduced by means of copying the structure of related components. Based on the general structure of the component, the appropriate values for that particular instance are to be set. However, it regularly occurs that some values are forgotten and are left to the pasted value. Such errors are often referred to as copy-and-paste errors.

Suppose that for a system with two TRSs it is observed, during simulation, that the controller does not comply to the requirements. However, the exact reason cannot be derived by means of simulation. Hence, the error will be sought in the icon overview.

The first step is to check the dependencies at the highest level of abstraction as

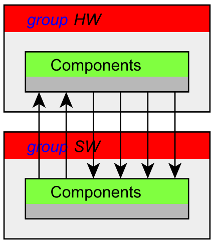


Figure 2.20: Dependencies between two TRSs at the highest level of abstraction.

depicted in Figure 2.20. This view shows the interface between the hardware (HW) and the software (SW) components. Intuitively, the antisymmetric nature of the interface between the software and the hardware seems odd. Namely, the software send one signal to the hardware (MTR) and the hardware one to the software (PEC). Hence, the number of interactions should be symmetric.

To investigate further, a less coarse view is chosen as depicted in Figure 2.21. In yEd this can be obtained by unfolding the hierarchy. Note that some relations have been removed for the sake of simplicity.

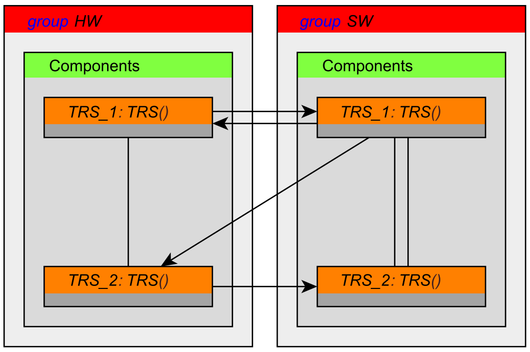


Figure 2.21: Less coarse view on the dependencies between two TRSs.

It can be observed that hardware component `TRS_2` has a dependency on the software component `TRS_1`. As the software component provides the motor signal to the hardware component, most likely the motor signal is wrongly assigned.

Finally, the most detailed view is depicted in Figure 2.22. Again, some relations have been removed for the sake of simplicity.

In this latest overview it can be clearly seen that the motor signal is wrongly coupled. Namely, the motor signal of the first TRS is coupled to the second TRS. This entails that if the motor of the first TRS is enabled, the motor of the second TRS is also enabled. This is not the behavior that is desired and is induced due to a copy-and-paste error. Based on these observations, the copy-and-paste error can be

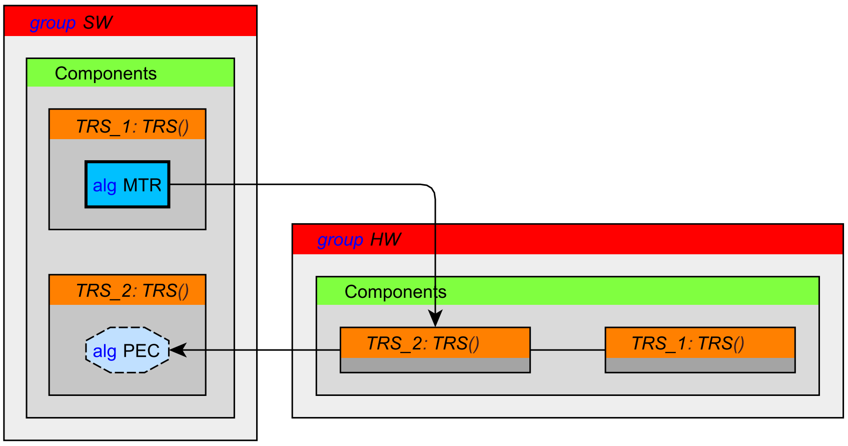


Figure 2.22: Detailed view on the dependencies between two TRSs.

solved accordingly. Again the quality of the system can be assessed in an early stage and fixing the errors can be done more easily.

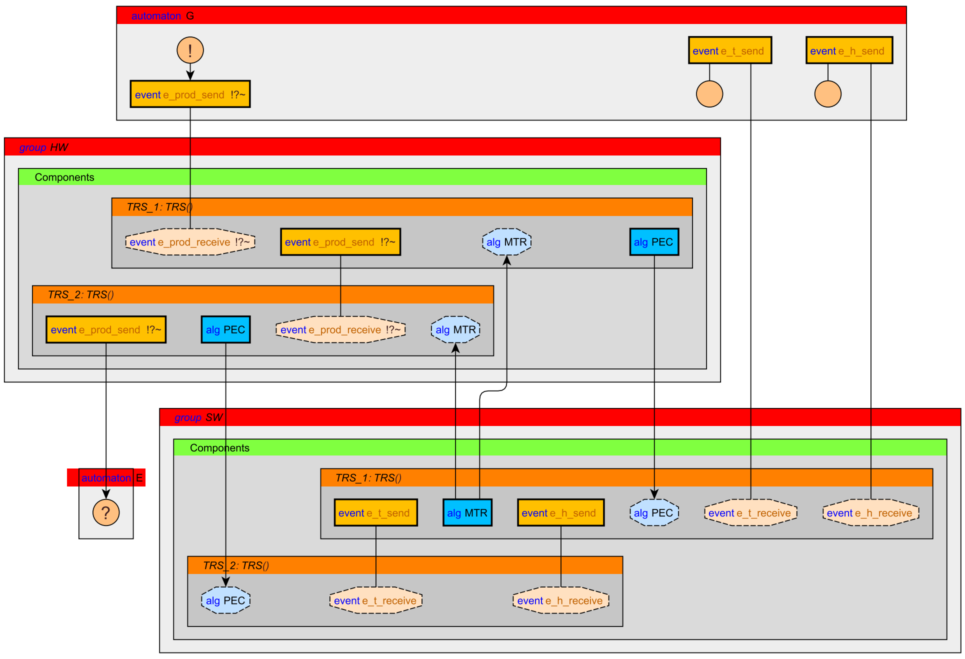


Figure 2.23: Detailed view on the dependencies between two TRSs, without abstraction.

In Figure 2.23, the detailed view without abstractions is provided. Although some of the names of the events are different, the structure of the system of Figure 2.15 and Figure 2.2 can be clearly seen. Note that the automata G and E are stubs, providing a source (generator) and a sink (exit).

2.5 Concluding remarks

The complexity of airport BHSs in combination with the required high level of robustness makes designing supervisory controllers for these systems a challenging task.

We have observed how the high-level modeling elements of the applied CIF MBD framework allows the modeler to concentrate on implementing the BHS design requirements, instead of programming specifically for the desired control platform. This way of requirement modeling made it relatively easy to pinpoint the causes of observed erroneous behavior in the simulation and visualization of the supervisory controller connected to the plant model. It also allowed a modular and hierarchical design of the supervisory controller, and provided flexibility in adapting and extending the model. In combination with the ability to model both the controller and the uncontrolled plant in the same modeling, simulation and validation environment, this has made it possible to catch modeling errors in the validation environment which most likely would have lead to errors later in the design process. As a result, this modeling approach will lead to a shorter modeling, testing and error correction iteration loops.

The results of this research project have been very much appreciated by Vanderlande, and have indeed inspired them to start MBD projects for supervisory control of BHSs.

Chapter 3

Supervisory control reasoning

This chapter is dedicated to supervisory controller synthesis and the derivation of user-feedback when synthesis fails to provide a desired result using so-called proof trees. A sound and complete framework is introduced which is the basis for an algorithm that derives a subset of all possible reasons for the absence of states.

3.1 Introduction

A first attempt in providing user-feedback when synthesis fails to provide a desired result was given by Swartjes et al. (2016). Information about blocking conditions is saved during synthesis in the supervisor using colors to encode the different requirements. A cause is then a trace that is decorated with colors, from the initial state to the state under investigation showing which states violated which requirements. In contrast to this first attempt, this chapter has a formal foundation and it is proven that the resulting feedback is "correct".

3.1.1 Supervisory control

What Many systems consist of multiple components that together must perform a specific task. A so-called *supervisory controller* (Cassandras and Lafortune, 2009) coordinates the components by disabling a specific set of actions that lead to undesired behavior in each state. A supervisory controller can be either created manually or can be synthesized. In this chapter, contrary to the previous chapter, the synthesized kind is investigated.

How Given a model of the uncontrolled system, i.e., the plant, and a model of the specifications, i.e., the requirements, synthesis creates a supervisor, i.e., a model of the supervisory controller. Synthesis disables actions in the plant such that safeness, nonblockingness, and controllability are enforced for all reachable states of the supervised system. In essence, the supervised system always abides to the requirements, guarantees that always a desired state can be reached from the initial state(s), and never blocks actions that cannot be controlled by the system.

Specifics Synthesis exists in many flavors. Synthesis was originally developed for automata without variables, e.g., (Ramadge and Wonham, 1987). Later, synthesis was extended to automata that contain variables, e.g., (Ouedraogo et al., 2010). In addition, three ways of specifying requirements for synthesis exist. Namely, event-based requirements (Cassandras and Lafortune, 2009), state-based requirements (Ma and Wonham, 2006), and a hybrid variant (Markovski et al., 2010b). In this thesis, data is not considered and the requirements are state-based. This entails that a set of states of the plant is defined that may not be reached in the supervised system, in order to obtain safe behavior. Note that requirements (or specifications) provided in the form of automata can be transformed to a set of forbidden states following Malik and Flordal (2008).

3.1.2 Lack of adoption

Although synthesizing supervisors provides a seemingly effortless approach to the creation of safe and nonblocking controllers, this technique is not (yet) fully embraced in industry. Only a few examples exist in literature in which synthesis is used for (small) industrial cases (Vahidi et al., 2006; Forschelen et al., 2012; Theunissen et al., 2014; Reijnen et al., 2017). Two main reasons can be pinpointed for this low adoption, as observed by the authors.

Scalability Firstly, scalability is an issue. For industrial-sized systems, both memory problems and long computation times may be encountered. Improvements have been made in the last decade to improve scalability (Ma and Wonham, 2006; Vahidi et al., 2006; Mohajerani et al., 2014; Fei et al., 2014; Miremadi and Lennartson, 2016). Nevertheless, scaling still remains a problem that must be tackled before any industrial adoption may be achieved.

Reasoning Secondly, the current algorithms and tools are not dedicated to users in industry. This is also observed by Wonham et al. (2017). For example, current implementations of synthesis algorithms lack the capabilities of providing feedback on the resulting supervisor. During the early stages of the supervisory controller design, desired behavior may be unexpectedly excluded from the supervised (controlled) system, or, even worse, the synthesis algorithm frequently cannot return a supervisor that abides to the requirements of the system. Providing feedback or reasons that explain why a supervised system cannot be synthesized or why certain behavior is absent are not provided. Currently, finding reasons demands knowledge on the synthesis algorithm which is normally nonexistent in industry: it requires working out long chains of steps in which states are successively excluded. Even for experts, deriving reasons for the absence of a specific state may be too demanding.

3.1.3 Contribution

As scalability is not the sole issue faced in the adoption of synthesis in industry, it is important to also create a proper basis for a providing reasons on the absence of states from a supervised controller. Therefore, our contribution is on the creation of a framework for the derivation of those reasons. Implementation of the proposed framework will naturally suffer from the same scalability issues as any supervisory control synthesis algorithm. Nevertheless, the framework introduced in this thesis will provide methods to derive the absence of specific states in the supervisor and provide insights on further extensions of providing user-feedback for synthesis. Investigating scalability of the proposed framework is future work.

The following contributions are made in this chapter:

- The definition of a cause and its correctness,
- the introduction of a deduction system for the derivation of causes for states removed during synthesis,
- a proof on the soundness and completeness of this deduction system, and
- an adapted synthesis algorithm that provides a single cause for each state removed during synthesis.

3.2 Related work

The properties to which a supervisor should comply can be viewed upon as verification properties. Using verification techniques, a supervisor can be created as done in, for instance, Åkesson et al. (2002), Jiang and Kumar (2006), Claessen et al. (2009), and Ehlers et al. (2014).

Åkesson et al. (2002) allows for the generation of paths to states that violate the controllability property of the supervisor in a modular setup; it is verified for a modular setup of plants and requirements whether a state will impose controllability issues when constructing the full synchronous composition. A path may be presented, if present, from the initial state to a state violating the controllability property. Returning this path may provide the user insight into how this state is reached and, therefore, how the plant must be adapted to prevent this state from being reached. This may provide insights on controllability of that particular state, but it does not explain the absence of a generic state. Moreover, a path is returned which does not contain all information on the branching nature of the system.

Claessen et al. (2009) introduces a method for synthesis based on SAT-solvers. The plant and some synthesis goals, i.e., safeness and controllability, are encoded as SAT-formulas and fed to a solver. This is done iteratively, starting from a formula only incorporating the initial state, incorporating more transitions along the way. The counterexamples obtained from the solver are used to adjust the system until no counterexamples can be found anymore; the resulting system is the supervisor. Although nonblockingness is not taken into account, the method allows for some insight in the effects of synthesis. However, it does not give insights in the absence of an arbitrary state.

Jiang and Kumar (2006) and Ehlers et al. (2014) both provide insights on how to encode several properties as verification formulas, but do not give insight on how a counterexample can be used for the refinement of the plant. As stated in the introduction, our aim is to generate a reason for the omission of a specific state from a supervisor. A characteristic of standard verification is that one or multiple traces, i.e., counterexample(s), are provided to a state violating the property under investigation. However, this state is not necessarily the state for which a reason is desired. Therefore, a lot of work must still be done to investigate the reason of absence of the state under investigation; it must be determined how the path and state of the counterexample is related to the state under investigation.

Chechik and Gurfinkel (2007) introduces a method of generating proof-like counterexamples or evidence. A more general version of providing explainable evidence is given by Cranen et al. (2015). In essence, instead of giving one or multiple traces to a state that violates the verification property, a tree is provided which explains the violation (step-by-step). This tree allows for storing information on the branching nature of the system. Using such a method, it is clearer why a state violates the property. A similar approach will be used in this thesis; proof-like evidence will be the base of the cause.

Markovski et al. (2010b) defines an unified framework for synthesis and verification, allowing for the specification of *desired* states. Using a verification tool, counterexamples can be derived for the absence of such a desired state. However, these counterexamples do not provide insight in the synthesis; a succeeding state can be pinpointed as the reason of absence, but it cannot be (directly) derived why that specific state is absent.

3.3 Definitions

In this section, formal definitions are introduced needed to provide and prove the complete and sound derivation of causes. First, a formal definition of a finite state automaton is introduced resembling the systems of Keller (1976); Alur (1999); Henzinger (2000). Next, the definitions needed for supervisory control are introduced. Finally, definitions are introduced for the creation of proof trees.

3.3.1 Finite state automata

Finite state automata are used to model the uncontrolled system, i.e., the plant, and the supervised system, i.e., the supervisor.

An automaton consists of states, modeling the different modes of the system. A transition models the change of state, captured by a transition relation. To distinguish between the different transitions, the transitions are labeled by means of events. To model the modes in which the system can start, the automaton contains a set of initial states. Additionally, a set of accepting, or marked, states is defined which define those states where the system should always be able to return to.

Definition 1 (Finite state automaton)**.** *An finite state automaton is defined as a quintuple $(X, \Sigma, \longrightarrow, X_0, X_m)$ with X the set of states, Σ the set of events, $\longrightarrow \subseteq X \times \Sigma \times X$ the transition relation, $X_0 \subseteq X$ the set of initial states, and $X_m \subseteq X$ the set of marked states.*

The notation $x \xrightarrow{\sigma} x'$ is used to denote that $(x, \sigma, x') \in \longrightarrow$. The notation $x \longrightarrow x'$ is used to denote that there exists a $\sigma \in \Sigma$ such that $x \xrightarrow{\sigma} x'$. Additionally, $\longrightarrow$ is naturally extended to its reflexive transitive closure $\xrightarrow{\star} \subseteq X \times X$. Dually, $x \xrightarrow{\star}$ is used to denote the set of all states reachable from x, and $X \xrightarrow{\star} = \bigcup_{x \in X} x \xrightarrow{\star}$.

When synthesizing controllers, it is (sometimes) necessary to remove states from the automaton. Removing states will result in a subsystem (or subgraph).

Definition 2 (Subsystem)**.** *Given two automata*

$$t_1 = (X_1, \Sigma, \longrightarrow_1, X_{0,1}, X_{m,1}), \textit{ and } t_2 = (X_2, \Sigma, \longrightarrow_2, X_{0,2}, X_{m,2})$$

t_2 is a subsystem of t_1, denoted by $t_2 \preceq t_1$, if, and only if,

$$\begin{array}{rcl} X_2 & \subseteq & X_1 \\ \longrightarrow_2 & = & \longrightarrow_1 \cap (X_2 \times \Sigma \times X_2) \\ X_{0,2} & = & X_{0,1} \cap X_2 \\ X_{m,2} & = & X_{m,1} \cap X_2 \end{array}$$

3.3.2 Paths

The notion of a path is necessary for the generation of causes.

Definition 3 (Path)**.** *Given an automaton $t = (X, \Sigma, \longrightarrow, X_0, X_m)$, a path from $x \in X$ to $x' \in X$ is a finite sequence of n states $[x_i]_{1 \leq i \leq n}$ such that*

- $x_1 = x$,
- $\forall 1 \leq i < n : x_i \longrightarrow x_{i+1}$,
- $x_n = x'$

The set of all paths from $x \in X$ to $x' \in X$ is denoted as $P(x, x')$. This is naturally extended, such that $P(X, x') = \bigcup_{x \in X} P(x, x')$ and $P(x, X') = \bigcup_{x' \in X'} P(x, x')$.

A path $p = [x_1 x_2 x_3]$ is henceforth visualized as $x_1 \longrightarrow x_2 \longrightarrow x_3$. Moreover, the notation $x \in p$ is used to denote the presence of state x in path p.

3.3.3 Supervisory control

The goal of synthesis is to provide a supervisor such that in the system under supervision safeness, nonblockingness, and controllability are enforced on all reachable states. A state is reachable if the state can be reached from an initial state. It is assumed that in the plant initially all states are reachable.

As stated, different flavors of synthesis exists. In this chapter, state-based supervisory control is considered similar to (Vahidi et al., 2006). By choice, the supervisor is a subsystem of the plant and states and transitions are removed from the plant during synthesis, in order to obtain a supervisor. In this particular case, the supervisor and supervised system are equal. Finally, it is assumed that the resulting supervisor is trimmed. This entails that no unreachable states are present in the supervisor.

Definition 4 (Supervisor)**.** *Given an automaton $(X, \Sigma, \longrightarrow, X_0, X_m)$ and a set of forbidden states $X_f \subseteq X$, a supervisor is a subsystem that is safe, nonblocking, controllable, and all states are reachable.*

With safeness, the absence of behavior in an automaton that leads to undesired (or forbidden) states is meant. As only state-based requirements are considered, a set of forbidden states (X_f) is considered for the synthesis problem.

Definition 5 (Safeness). *Given an automaton $(X, \Sigma, \longrightarrow, X_0, X_m)$ and a set of forbidden states $X_f \subseteq X$, the automaton is safe if, and only if,*

$$X_0 \xrightarrow{\star} \cap X_f = \varnothing$$

Complementary, nonblockingness is the ability to reach a specific state, i.e., a marked state. Hence, a system is nonblocking if from all reachable states, a marked state can be reached.

Definition 6 (Nonblocking). *Given an automaton $(X, \Sigma, \longrightarrow, X_0, X_m)$, the automaton is nonblocking if, and only if,*

$$\forall x \in X_0 \xrightarrow{\star} : x \xrightarrow{\star} \cap X_m \neq \varnothing$$

In supervisory control theory, a distinction is made between controllable and uncontrollable events. Controllable events can be blocked by the supervisor, while uncontrollable events cannot. Hence, if the plant contains a state from which an uncontrollable event is possible and the supervisor contains the same state, the supervisor must also be able to execute the uncontrollable event.

Definition 7 (Controllability). *Given two automata*

$$t_1 = (X_1, \Sigma, \longrightarrow_1, X_{0,1}, X_{m,1}), \textit{and } t_2 = (X_2, \Sigma, \longrightarrow_2, X_{0,2}, X_{m,2})$$

and a partitioning of their common alphabets into two disjoint sets representing the controllable events (Σ_c) and the uncontrollable event (Σ_u), subsystem t_2 is controllable with respect to t_1 if, and only if,

$$\forall x \in X_2, \forall \sigma \in \Sigma_u : x \xrightarrow{\sigma}_1 \implies x \xrightarrow{\sigma}_2$$

If subsystem t_2 is controllable with respect to t_1, t_2 is called a controllable subsystem of t_1 or, simply, controllable.

Without nonblockingness and controllability, a supervisor can be created by simply removing the forbidden states (and involved transitions) from the plant. However, the removal of these states may infringe the controllability and nonblockingness of the supervisor. Synthesis is the process of creating the largest supervisor that is safe, nonblocking, and controllable. A supervisor s_1 is larger than s_2, i.e., allows for more behavior; fewer states have been removed to create a safe, nonblocking, and a controllable supervisor. Note that the largest supervisor may be empty, when safeness, nonblockingness and controllability cannot be enforced without removing the initial states.

3.3.4 Proof trees

In this thesis, proofs are created to explain the absence of specific states in the supervisor, i.e., to provide causes. This is done by means of deduction rules and so-called proof trees, for which the formal definitions can be found in Mousavi et al. (2007).

A deduction rule is a pair, consisting of a set of premises P, and a conclusion c. The conclusions and premises are predicates expressions over the states of the automaton.

An example of such an expression is reach(x) that denotes the reachability of state x. The conclusion is bound by implication; if all the premises hold, then the conclusion holds.

Deduction rules are depicted as follows:

$$\frac{x' \longrightarrow x \quad \text{reach}(x')}{\text{reach}(x)}$$

Zero or more premises are given above the line, with the conclusion below the line. This particular deduction rule specifies that if there is a transition from x' to x and x' is reachable, then reach(x) holds (or x is reachable).

Multiple rules together make a deduction system. To distinguish between the multiple rules, a name is provided at the right-hand side of the rules. As an example, consider the following deduction system:

$$\frac{x' \longrightarrow x \quad \text{reach}(x')}{\text{reach}(x)}\,(1) \qquad \frac{x \in X_0}{\text{reach}(x)}\,(2)$$

From the set of rules, a proof is made by subsequently applying deduction rules until a trivially true condition is obtained. Suppose, it is desired to prove the property reach(x_2) with the knowledge that $x_0 \longrightarrow x_1 \longrightarrow x_2$ and $x_0 \in X_0$. The proof for this particular property is given by means of a tree:

$$\frac{x_1 \longrightarrow x_2 \qquad \dfrac{x_0 \longrightarrow x_1 \qquad \dfrac{x_0 \in X_0}{\text{reach}(x_0)}\,(2)}{\text{reach}(x_1)}\,(1)}{\text{reach}(x_2)}\,(1)$$

The labels on the right-hand side denote the applied rule.

Definition 8 (Proof)**.** *Given a conclusion c, a proof for c is a well-founded upwardly branching tree with c as the root of the tree and each formula and its predecessors in the tree form an instance of one of the deduction rules. The resulting tree is called the proof tree.*

The notation $\vdash c$ is used to denote that there is a proof (tree) with a conclusion c. Note that multiple proofs can exists for the same conclusion c.

3.4 Deduction system

In this section, the deduction rules for the absence of states in the supervisor are defined. This is done in a piecewise manner, introducing new rules as needed. All rules are defined on the automaton modeling the plant. It is assumed that the set of forbidden states X_f is defined. The predicates gone(x) is defined to denote the absence of a state from the resulting supervisor. A proof tree with gone(x) as a conclusion explains the absence of that state.

To explain why these causes are reasonable, consider the plant in Figure 3.1. State x_2 is unsafe and hence not present in the supervisor. As a consequence, it is

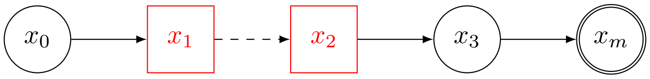


Figure 3.1: Small system with both x_1 and x_2 bad states.

not difficult to see that states x_3 and x_m are no longer reachable. Therefore, these states will also no longer be present in the resulting supervisor.

Based on the definition of a supervisor, i.e., Definition 4, four reasons can be derived for the absence of a state in the supervisor. A state is absent, if it is unsafe, has controllability issues, is blocking or is unreachable. These four reasons are considered in the remainder of this section.

Safeness

Based on the definition of a safe automaton, i.e., safeness, it can be easily derived when a state is considered unsafe; a state is unsafe if it is a forbidden state. Any unsafe state needs to be removed during synthesis. This is expressed by:

$$\frac{x \in X_f}{\text{gone}(x)} \text{ (s)}$$

3.4.1 Controllability

Only the execution of controllable events may be prevented by the supervisor. As a result, a state needs to be removed if an unsafe state can be reached with an uncontrollable transition.

$$\frac{x \dashrightarrow x' \quad \text{gone}(x')}{\text{gone}(x)} \text{ (c)}$$

Here, $\dashrightarrow$ is used to denote the presence of an uncontrollable transition, i.e., a transition with an uncontrollable event.

Blockingness

To derive whether a state is blocking is somewhat more involved. Namely, for a state x to be blocking, there must be no path leading from that state to a marked state. This implies that either there were never such paths, or that each of these paths, as there can be multiple, contains at least one bad state; if a single path from x to a marked state, e.g., x_m, contains a bad state, x cannot reach x_m via that specific path.

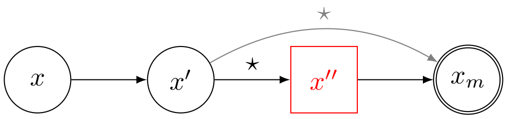


Figure 3.2: Schematic overview of a (non)blocking situation, where x'' is a bad state.

Schematically, this is depicted in Figure 3.2 where state x'' is a bad state. Not considering the gray path at the moment, the marked state x_m cannot be reached.

However, if the gray path is present in the system, it is clear that the marked state is still reachable. Hence, a state x is blocking if all paths from x to a marked state contain a bad state.

$$\frac{\forall p \in P(x, X_m) : \exists x' \in p : \text{gone}(x')}{\text{gone}(x)} \text{ (blk)}$$

First-order logic is used in the above rule. In essence, the quantifiers (existential and universal) result in a logical or (creating multiple trees), respectively, logical and. The formal implications of these operators on a tree have been investigated in the work of van Weerdenburg and Reniers (2009). For the sake of simplicity an alternative form of the above rule will be used for the remainder of this chapter in which the existential and universal quantifiers are removed:

$$\frac{\text{gone}(x_1) \quad \cdots \quad \text{gone}(x_n)}{\text{gone}(x)} \text{ (blk)}$$

for each set $\{x_1, \cdots, x_n\} \subseteq X$ such that $\{x_1, \cdots, x_n\} \cap p \neq \varnothing$ for any $p \in P(x, X_m)$.

3.4.2 Reachability

The reachability of a state can change as a consequence of synthesis. In Figure 3.3, for example, both states x_1 and x_3 will become unreachable as a consequence of synthesis; state x_0 will be removed as there is an uncontrollable transition possible to the unsafe state x_2.

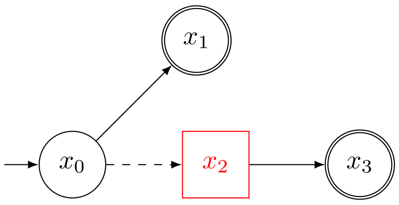


Figure 3.3: Unreachability of states x_1 and x_3, as a consequence of synthesis.

Similarly to a blocking state, a state x is (or has become) unreachable if all paths leading from an initial state to state x contain a bad state.

$$\frac{\text{gone}(x_1) \quad \cdots \quad \text{gone}(x_n)}{\text{gone}(x)} \text{ (r)}$$

for each set $\{x_1, \cdots, x_n\} \subseteq X$ such that $\{x_1, \cdots, x_n\} \cap p \neq \varnothing$ for any $p \in P(X_0, x)$.

3.4.3 Summary

A summary of all rules and a short description is provided in Table 3.1.

Table 3.1: Summary of all deduction rules.

Rule	Description
$\dfrac{x \in X_f}{\text{gone}(x)}$ (s)	If a state is forbidden, the state is unsafe
$\dfrac{x \dashrightarrow x' \quad \text{gone}(x')}{\text{gone}(x)}$ (c)	When a state x' is proven to be unsafe and that state can be reached from state x by means of an uncontrollable transition, the state is declared bad
$\dfrac{\text{gone}(x_1) \quad \cdots \quad \text{gone}(x_n)}{\text{gone}(x)}$ (blk) for each set $\{x_1, \cdots, x_n\} \subseteq X$ such that $\{x_1, \cdots, x_n\} \cap p \neq \varnothing$ for any $p \in P(x, X_m)$	When no path to a marked state can be found, that state is blocking
$\dfrac{\text{gone}(x_1) \quad \cdots \quad \text{gone}(x_n)}{\text{gone}(x)}$ (r) for each set $\{x_1, \cdots, x_n\} \subseteq X$ such that $\{x_1, \cdots, x_n\} \cap p \neq \varnothing$ for any $p \in P(X_0, x)$	When no path from an initial state can be found, that state is unreachable

3.5 Soundness

In this section, we provide a theorem that defines that a state x, for which we can provide a proof of absence (gone), is actually absent from the supervised system. Or, we can never provide a proof of absence of a state x that is present in the supervised system.

Theorem 1. *Given an automaton $(X_p, \varSigma, \longrightarrow_p, X_{0,p}, X_{m,p})$, a set of forbidden states X_f and the maximally permissive supervisor $(X_s, \varSigma, \longrightarrow_s, X_{0,s}, X_{m,s})$, it holds that*

$$\vdash \text{gone}(x) \implies x \notin X_s \tag{3.1}$$

Proof. The property is proven using induction on the depth of the proof tree for $\vdash \text{gone}(x)$. Suppose that $\vdash \text{gone}(x)$. It needs to be shown that $x \notin X_s$. We use a case distinction on the deduction rule applied last in the proof of $\vdash \text{gone}(x)$.

(i) The deduction rule applied last is (s). Then it must hold that $x \in X_f$. From $x \in X_f$ it follows immediately that $x \notin X_s$, since the supervisor is safe (and all supervisor states are reachable).

(ii) The deduction rule applied last is (c). Then $\vdash \text{gone}(x')$ for some x' such that $x \dashrightarrow x'$. By induction, as the proof tree for $\vdash \text{gone}(x')$ is smaller than the proof tree for $\vdash \text{gone}(x)$, we have that $x' \notin X_s$. In case $x \in X_s$ this would mean that the supervisor is not controllable.

(iii) The deduction rule applied last is (blk). Then it must be the case that

$$\frac{\text{gone}(x_1) \quad \cdots \quad \text{gone}(x_n)}{\text{gone}(x)}$$

for some set $X' = \{x_1, \cdots, x_n\} \subseteq X$ such that $\{x_1, \cdots, x_n\} \cap p \neq \varnothing$ for any $p \in P(x, X_m)$.

We have $\vdash \text{gone}(x')$ for all $x' \in X'$. The proof trees for $\vdash \text{gone}(x')$ for all $x' \in X'$ are smaller than the proof tree for $\vdash \text{gone}(x)$ and hence, by induction, we have that $x' \notin X_s$. This means that p is not a path from x to a marked state in the supervisor. As this holds for all paths $p \in P(x, X_m)$ and the supervisor is a subsystem of the automaton, there is no path in the supervisor from x to a marked state and hence the supervisor would be blocking in case $x \in X_s$. Therefore we can safely conclude that $x \notin X_s$.

(iv) The deduction rule applied last is (r). Then it must be the case that

$$\frac{\text{gone}(x_1) \quad \cdots \quad \text{gone}(x_n)}{\text{gone}(x)}$$

for some set $X' = \{x_1, \cdots, x_n\} \subseteq X$ such that $\{x_1, \cdots, x_n\} \cap p \neq \varnothing$ for any $p \in P(X_0, x)$.

We have $\vdash \text{gone}(x')$ for all $x' \in X'$. By induction, as the proof tree for $\vdash \text{gone}(x')$ for all $x' \in X'$ is smaller than the proof tree for $\vdash \text{gone}(x)$, we have that $x' \notin X_s$. Hence, it has to be the case that also $x \notin X_s$ as there is no path left from any initial state to x and all states in the supervisor are reachable.

□

3.6 Completeness

Complementary to soundness, the proof system is complete if each absent state has (at least) one associated clause.

The cyclic dependencies of the rules prove difficult to show completeness. As an example, consider a state x. To prove that state x is unreachable, a proof is needed using rule (r). This entails that we need to proof gone for a set of predecessor states. With exception of the rule (s), proving gone for such a predecessor state entails that we must prove that its successor states are gone for which the proof gone(x) is needed (which we are trying to prove).

To break these cyclic dependencies, completeness is proven with the aid of an iterative definition of synthesis. During each iteration of the algorithm, states that lead to undesired behavior are identified and removed from the supervisor.

The slightly adjusted algorithm of Ouedraogo et al. (2010) is used; the algorithm is adjusted to also take the reachability of states after synthesis into consideration. As each iteration new states may be identified and the algorithm finishes in a finite number of iterations, induction can be used to create the completeness proof.

```
**Require:** A plant $p = (X_p, \Sigma, \longrightarrow_p, X_{0,p}, X_{m,p})$
**Require:** Set of forbidden states $X_f$
**Ensure:** A supervisor $s := (Y, \Sigma, \longrightarrow \cap (Y \times \Sigma \times Y), X_{p,0} \cap Y, X_{p,m} \cap Y)$
 1: $k := 0, X^k := X_p$
 2: **repeat**
 3:     $i := 0, j := 0$
 4:     Initialize the set of nonblocking states, $N_0^k := X_m \cap X^k$
 5:     **repeat**
 6:         $N_{i+1}^k := N_i^k \cup \{x \in X^k \mid x \longrightarrow y, y \in N_i^k\}$
 7:         $i := i + 1$
 8:     **until** $N_i^k = N_{i-1}^k$
 9:     $N^k := N_i^k$
10:     Initialize the set of blocking states, $B_0^k := (X_f \cap X^k) \cup (X^k \setminus N^k)$
11:     **repeat**
12:         $B_{j+1}^k := B_j^k \cup \{x \in X^k \mid x \dashrightarrow y, y \in B_j^k\}$
13:         $j := j + 1$
14:     **until** $B_j^k = B_{j-1}^k$
15:     $B^k := B_j^k$
16:     Remove all blocking states, i.e., $X^{k+1} = X^k \setminus B^k$
17:     $k := k + 1$
18: **until** $X^k = X^{k-1}$
19: $i := 0$
20: Initialize the set of reachable states, $Y_0 := X_0 \cap X^k$
21: **repeat**
22:     $Y_{i+1} := Y_i \cup \{x \in X^k \mid y \longrightarrow x, y \in Y_i\}$
23:     $i := i + 1$
24: **until** $Y_i = Y_{i-1}$
25: $Y := Y_i$
```

Note that the last steps determining the reachability of states are an addition to the algorithm. This is done to coincide with our definition of a proper supervisor. The actual synthesis is not influenced by the addition of these steps and, clearly, only states that cannot be reached from an initial state are additionally removed.

To aid the proof of completeness, a handful of lemmas are introduced. Firstly, it is proven that each absence of a state from the supervised system is due to a blocking state or a change in reachability due to the removal of states during synthesis.

Lemma 1. *Given a plant $p = (X_p, \Sigma, \longrightarrow_p, X_{0,p}, X_{m,p})$ and a synthesized supervisor $s = (X_s, \Sigma, \longrightarrow_s, X_{0,s}, X_{m,s})$. For any $x \in X_p \setminus X_s$, it holds that $x \in B^i$ for some $i \in \mathbb{N}$, or $x \in X^k$ and $x \notin Y$ with k the final iterator of the synthesis algorithm.*

Proof. During synthesis, the set of states is pruned based on the set of blocking states (see line 16). As a result it must hold that if $x \in X_p$ but not in X^k with k the final iterator of the algorithm, the state must have been removed at iteration i based on the fact that $x \in B^i$ for some $i \in \mathbb{N}$.

When $x \in X_p$ and $X \in X^k$ but $x \notin Y$ with k the final iterator, it must hold (according to line 22), that there is no path from the pruned set of initial states to state x.

Hence, if $x \notin X_s$ either there is an iterator i when the state becomes blocking or the state was removed because it is no longer reachable. □

Secondly, it is proven that if during any iteration it is derived that a state is not contained in the set of nonblocking states, then from this state no path exists to a marked state.

Lemma 2. *Given a plant $p = (X_p, \Sigma, \longrightarrow_p, X_{0,p}, X_{m,p})$. For any $x \in X_p$, if $x \notin N^j$ for some $j < k$ with k the final iterator of the synthesis algorithm, it must hold that $P(x, X_m \cap X^j) = \varnothing$.*

Proof. Using the fact that the set of nonblocking states is increased each iteration as $N^j_{i+1} := N^j_i \cup \{x \mid x \longrightarrow y, y \in N^j_i\}$ (see line 6), it must hold that $N^j \supseteq N^j_i \supseteq X_m \cap X^j$. Hence, if $x \notin N^j$ it must hold that $x \notin N^j_i$ for all $i \in \mathbb{N}$. As a result, there is no path to a marked state as otherwise the state would have been nonblocking. This can be generally extended to any j, such that $x \notin N^j \implies P(x, X_m \cap X^j) = \varnothing$ for all $j \in \mathbb{N}$. □

Thirdly, a similar result can be obtained for the reachability of states; there exists no path from an initial state to a state that is not contained in the set of reachable states.

Lemma 3. *Given a plant $p = (X, \Sigma, \longrightarrow, X_0, X_m)$, if $x \in X^k$ and $x \notin Y$ it must hold that $P(X_0 \cap X^k, x) = \varnothing$ with k the final iterator of synthesis.*

Proof. Likewise, Lemma 2. □

Finally, completeness is proven by combining the above three lemmas.

Theorem 2. *Given an plant $(X_p, \Sigma, \longrightarrow_p, X_{0,p}, X_{m,p})$, a set of forbidden states X_f and the maximally permissive supervisor $(X_s, \Sigma, \longrightarrow_s, X_{0,s}, X_{m,s})$, it holds that*

$$x \notin X_s \implies \vdash \text{gone}(x) \tag{3.2}$$

Proof. The property is proven using the introduced three lemmas. When $x \in X_p$ and $x \notin X_s$ it must hold, following Lemma 1, that there exists some i such that $x \in B^i$ or $x \in X^k$ and $x \notin Y$ with k the final iterator of the algorithm. The former is firstly investigated, as Y depends on the results of successively removing states from X_p. The existence of causes is proved by means of induction to the iteration in which the state was removed.

- Assume that at iteration $k = 0$ the state x is trivially blocking, i.e., $x \in B^0$. As B^0 is determined by means of a fixpoint calculation, induction will also be used for the iterations of this predicate.
  - Assume that state x is included in B^0_0. It must trivially hold that $x \in X_f$. Hence, the cause
$$\frac{x \in X_f}{\text{gone}(x)}$$
    can be derived for state x.

  - Assume that state x is added to B^0_{j+1} (so it was not included in B^0_j). By induction we know that causes can be created for all states $y \in B^0_j$. It must be proven that a cause can be generated for $x \in B^0_{j+1}$. From the algorithm it is known that $B^0_{j+1} := B^0_j \cup \{x \in X^0 \mid x \dashrightarrow y, y \in B^0_j\}$, and it is known that $x \notin B^0_j$ as else there would be not fixpoint. Hence, $\exists x \in X_0 : x \dashrightarrow y, y \in B^0_j$ with

    $$\frac{x \dashrightarrow y \quad \text{gone}(y)}{\text{gone}(x)}$$

    as the cause. Using the induction hypothesis, it is known that a cause already exists for gone(y).

- Assume that at iteration $k > 0$ the state x is trivially blocking, i.e., $x \in B^k$. By induction we know that causes can be created for all states $y \in B^{k-1}$. Note that $y \in B^{k-1}$ entails that $y \in X_p$ and $y \notin X_s$ or, in other words, gone(y).

  - Assume that state x is included in B^k_0. It must hold that $x \notin N^k$. Using Lemma 2 it is known that there does not exist a path to a marked state in the set $X_m \cap X^{k-1}$. Hence,

    $$\frac{\text{gone}(x_1) \quad \cdots \quad \text{gone}(x_n)}{\text{gone}(x)}$$

    for some set $\{x_1, \cdots, x_n\} \subseteq X$ such that $\{x_1, \cdots, x_n\} \cap p \neq \varnothing$ for any $p \in P(x, X_m \cap X^{k-1})$ is a cause for state x. Using the induction hypothesis, gone(x_1) through gone(x_n) exists, as that state must have been removed in an earlier iteration.

  - Assume that state x is added to B^k_{j+1} (so it was not included in B^k_j). By induction we know that causes can be created for all states $y \in B^k_j$. Following earlier derivations, again, the cause

    $$\frac{x \dashrightarrow y \quad \text{gone}(y)}{\text{gone}(x)}$$

    can be generated. Using the induction hypothesis, it is known that gone(y) exists.

Finally, the existence of a cause is concluded by considering the reachability of the states. Using Lemma 3, a state is removed when no path exists from the pruned set of initial states to the state under investigation. Hence,

$$\frac{\text{gone}(x_1) \quad \cdots \quad \text{gone}(x_n)}{\text{gone}(x)}$$

for some set $\{x_1, \cdots, x_n\} \subseteq X$ such that $\{x_1, \cdots, x_n\} \cap p \neq \varnothing$ for any $p \in P(X_0 \cap X^k, x)$ for which gone(x_1) through gone(x_n) are known to exist as a cause exists for every state removed during the previous stage of the synthesis process. $\square$

3.7 Extension

When considering the completeness proof, it is not difficult to see that a cause for the absence of state may be derived during synthesis. In this section, the synthesis algorithm is adapted to allow for the generation of these causes. The causes generated during synthesis do not necessarily provide all reasons.

```
Require: A plant $p = (X_p, \Sigma, \longrightarrow_p, X_{0,p}, X_{m,p})$
Require: Set of forbidden states $X_f$
Ensure: A supervisor $s := (Y, \Sigma, \longrightarrow \cap (Y \times \Sigma \times Y), X_{p,0} \cap Y, X_{p,m} \cap Y)$
Ensure: Mapping of states to causes $\kappa$
 1: $k := 0, X^k := X_p$
 2: repeat
 3:     $i := 0, j := 0$
 4:     Initialize the set of nonblocking states, $N_0^k := X_m \cap X^k$
 5:     repeat
 6:         $N_{i+1}^k := N_i^k \cup \{x \in X^k \mid x \longrightarrow y, y \in N_i^k\}$
 7:         $i := i + 1$
 8:     until $N_i^k = N_{i-1}^k$
 9:     $N^k := N_i^k$
10:     Initialize the set of blocking states, $B_0^k := (X_f \cap X^k) \cup (X^k \setminus N^k)$
11:     for $x \in B_0^k \cap X_f$ do $\kappa(x) = \dfrac{x \in X_f}{\text{gone}(x)}$ (s)
12:     for $x \in B_0^k \setminus X_f$ do $\kappa(x) = \dfrac{\kappa(x_1) \ \cdots \ \kappa(x_n)}{\text{gone}(x)}$ (blk) with $\{x_1, \cdots, x_n\} = \{y \in X \setminus \{x\} \mid x \longrightarrow y\}$
13:     repeat
14:         $B_{j+1}^k := B_j^k \cup \{x \in X^k \mid x \dashrightarrow y, y \in B_j^k\}$
15:         for $x \in B_{j+1}^k \setminus B_j^k$ do $\kappa(x) = \dfrac{x \dashrightarrow y \quad \text{gone}(x)}{\text{gone}(x)}$ (c) for some $y \in B_k^k : x \dashrightarrow y$
16:         $j := j + 1$
17:     until $B_j^k = B_{j-1}^k$
18:     $B^k := B_j^k$
19:     Remove all blocking states, i.e., $X^{k+1} = X^k \setminus B^k$
20:     $k := k + 1$
21: until $X^k = X^{k-1}$
22: $i := 0$
23: Initialize the set of reachable states, $Y_0 := X_0 \cap X^k$
24: repeat
25:     $Y_{i+1} := Y_i \cup \{x \in X^k \mid y \longrightarrow x, y \in Y_i\}$
26:     $i := i + 1$
27: until $Y_i = Y_{i-1}$
28: $Y := Y_i$
29: for $x \in X^k \setminus Y$ do $\kappa(x) := \dfrac{\kappa(x_1) \ \cdots \ \kappa(x_n)}{\text{gone}(x)}$ (r) with $\{x_1, \cdots, x_n\} = \{y \in X \setminus \{x\} \mid y \longrightarrow x\}$
```

Note that when κ is used in the right-hand side of the equation, the tree defined by κ is used. The tree is therefore only complete when the synthesis algorithm has finished.

Reviewing the provided algorithm, it is clear that the addition of the derivation of causes does not influence the synthesis algorithm. As a result, properties of the algorithm, like termination, are still valid. Note that the additions increase the complexity of the algorithm. However, the complexity of the synthesis part is not increased; the increase of complexity is solely due to the creation of the causes.

3.8 Examples

Using a standard synthesis algorithm, no supervisor can be computed for the system of Figure 3.3. Therefore, it must be possible to determine a cause for the absence of each of the states. For the state x_1 a cause can be derived, using the deduction system defined earlier:

$$\dfrac{\dfrac{x_0 \dashrightarrow x_2 \qquad \dfrac{x_2 \in X_f}{\text{gone}(x_2)}\,\text{(s)}}{\text{gone}(x_0)}\,\text{(c)}}{\text{gone}(x_1)}\,\text{(r)}$$

Alternative, the following cause may be derived:

$$\dfrac{\dfrac{\dfrac{\dfrac{x_0 \dashrightarrow x_2 \qquad \dfrac{x_2 \in X_f}{\text{gone}(x_2)}\,\text{(s)}}{\text{gone}(x_0)}\,\text{(c)}}{\text{gone}(x_1)}\,\text{(r)} \qquad \dfrac{\dfrac{x_2 \in X_f}{\text{gone}(x_2)}\,\text{(s)}}{\text{gone}(x_3)}\,\text{(r)}}{\text{gone}(x_0)}\,\text{(blk)}}{\text{gone}(x_1)}\,\text{(r)}$$

Note that infinitely many causes may be derived, as in this particular case, we can always add redundant parts to the cause where the intermediate conclusion relies on itself.

The cause(s) obtained from the extended synthesis algorithm do not have repetition (in depth) of the same conclusion. In this case, the first example cause provided in this section is obtained. Following the algorithm, $N_0^0 = \{x_1, x_3\}$ reaches the fixpoint $N^0 = \{x_0, x_1, x_2, x_3\}$ in the following two steps. Using the set of nonblocking states, the set of blocking states is determined as $B_0^0 = \{x_2\}$. This immediately gives rise to the first (partial) cause:

$$\kappa = \left\{ x_2 \mapsto \dfrac{x_2 \in X_f}{\text{gone}(x_2)}\,\text{(s)} \right\}$$

In the next step, the set of blocking states is extended with state x_0, i.e., $B_1^0 = \{x_0, x_2\}$. As a result,

$$\kappa = \left\{ x_0 \mapsto \dfrac{x_0 \dashrightarrow x_2 \quad \kappa(x_2)}{\text{gone}(x_0)}\,\text{(c)}, x_2 \mapsto \dfrac{x_2 \in X_f}{\text{gone}(x_2)}\,\text{(s)} \right\}$$

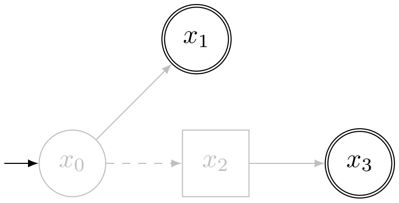


Figure 3.4: Unreachability of states x_1 and x_3, as a consequence of synthesis.

At the end of the first major iteration, states x_0 and x_2 are removed. This results in the subsystem depicted in Figure 3.4. As the remaining states are marked states, the synthesis processes finishes at $k = 1$ without removing any additional states. When the reachability of the states is assessed, it is immediately determined after the first iteration that both x_1 and x_3 are not reachable. As a result,

$$\kappa = \left\{ \begin{array}{lll} x_0 & \mapsto & \left\{ \dfrac{x_0 \dashrightarrow x_2 \quad \kappa(x_2)}{\text{gone}(x_0)}\ (\text{c}) \right\} \\ x_1 & \mapsto & \left\{ \dfrac{\kappa(x_0)}{\text{gone}(x_1)}\ (\text{r}) \right\} \\ x_2 & \mapsto & \left\{ \dfrac{x_2 \in X_f}{\text{gone}(x_2)}\ (\text{s}) \right\} \\ x_3 & \mapsto & \left\{ \dfrac{\kappa(x_2)}{\text{gone}(x_3)}\ (\text{r}) \right\} \end{array} \right\}$$

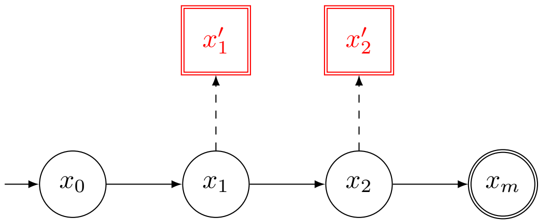


Figure 3.5: Example system in which multiple reasons for the absence of state x_1 can be derived.

In the example provided in Figure 3.5, multiple reasons can be derived for the absence of state x_1:

- The forbidden state x_1' can be reached by means of an uncontrollable transition, and
- x_1 is blocking as no path is present to the marked state x_m due to the removal of blocked state x_2, and
- x_1 has reachability problems due to the removal of x_0 as a consequence of the badness of state x_1 itself.

Using the adapted synthesis algorithm, omitting the intermediate synthesis steps, the following causes can be derived:

$$\kappa = \left\{ \begin{array}{lcc} x_0 & \mapsto & \left\{ \dfrac{\kappa(x_1)}{\text{gone}(x_0)}\ (\text{blk}) \right\} \\ x_1 & \mapsto & \left\{ \dfrac{x_1 \dashrightarrow x_1' \quad \kappa(x_1')}{\text{gone}(x_1)}\ (\text{c}) \right\} \\ x_1' & \mapsto & \left\{ \dfrac{x_1' \in X_f}{\text{gone}(x_1')}\ (\text{s}) \right\} \\ x_2 & \mapsto & \left\{ \dfrac{x_2 \dashrightarrow x_2' \quad \kappa(x_2')}{\text{gone}(x_2)}\ (\text{c}) \right\} \\ x_2' & \mapsto & \left\{ \dfrac{x_2' \in X_f}{\text{gone}(x_2')}\ (\text{s}) \right\} \\ x_m & \mapsto & \left\{ \dfrac{\kappa(x_2)}{\text{gone}(x_m)}\ (\text{r}) \right\} \end{array} \right\}$$

Clearly, the adapted algorithm can only derive a single, but sound, cause. To be able to derive multiple causes, another dedicated algorithm is needed. Implications and difficulties in creating such an algorithm are provided in Appendix A.

3.9 Concluding remarks

In this chapter, a deduction system is introduced for the derivation of causes. Causes are the proof trees associated with a proof of absence of a state from the supervisor; causes explain why a certain state is absent from the supervisor. It is proven that if a cause can be derived for a state, then the state is removed during synthesis, i.e., soundness. Complementary, with the aid of a synthesis algorithm, it is proven that if a state is removed during synthesis a cause can be derived, i.e., completeness.

Using the results of the completeness proof, an adapted synthesis algorithm has been introduced that allows for the generation of causes during the synthesis of the supervisor. Two examples are provided that show the derivation of causes with the adapted algorithm. Clearly, the proposed algorithm does not create all causes. In order to derive all causes, some challenges need to be overcome, as described in Appendix A.

In this chapter, requirements are provided in the form of forbidden states of the plant. Other forms of specifying requirements and other forms of requirements exist. For example, a lot of requirements are event-based, i.e., specifying which event can or cannot occur in a certain state. This type of requirements currently can be taken into account by using the method of Malik and Flordal (2008). In the future, it may be desired to extend the deduction rules to also consider the labels of the edges such that event-based requirements can be taken into account directly.

Although the proposed deduction system and the proposed algorithm suffer from the same scalability issues as the synthesis algorithm, providing an initial framework for the deduction of causes already shows a lot of potential: pinpointing the exact reason on the absence of a state reduces overhead of manually finding the reason and allows the modeler to understand the absence of the state. By showing how the synthesis algorithm fails to find a proper supervisor allows for a designated solution instead of an ad-hoc approach to make the synthesis work. In the end, this will improve the quality of the resulting supervisors while simultaneously reducing the effort.

Chapter 4

Relating transformed models

This chapter is dedicated to relating two models on a syntactical and semantical level. This is of importance when models need to undergo model-to-model transformations. The first part of the chapter is dedicated to the syntax or the traceability between two models, while the second part is dedicated to the semantics or the similarity of two models.

4.1 Introduction

Synchronization aids the modeler, as explained and shown in Chapter 2; synchronization helps to increase quality and flexibility while simultaneously decreasing model complexity. As long as synchronization is supported by the modeling formalism this is all fine. However, there are many modeling formalisms that do not support the concept of synchronization, as shown in this thesis. A commonly used modeling formalism that does not contain synchronization is Stateflow. There are, to the author, no known control platforms that support the concept of synchronization. If the created models are to be reused in different modeling formalisms or used for the purpose of code generation, a common approach is to eliminate synchronization[2].

Removal of synchronization is a known process and is often applied to networks of automata (Cassandras and Lafortune, 2009). A so-called synchronous product is computed from the network. The synchronous product, a single automaton, defines the complete synchronization-less behavior of the former network. As only a single automaton is now used to describe the former network of automata, it is trivial to see that the model structure changes heavily. To illustrate this change, a small model and its resulting synchronous product is shown in Figure 4.1. Note that the severity of the change depends on several factors and, worst-case, an exponential growth of the number of states of the system is observed.

What is immediately clear from Figure 4.1 is the duplication of locations and transitions containing non-shared events. Moreover, the individual specifications,

[2]Note that for the sake of code generation it is not necessary to remove synchronization, as shown by Park and Kim (1990). However, there are reasons why an alternative approach comprising the elimination of synchronization is preferred. This will be shown in Chapter 5.

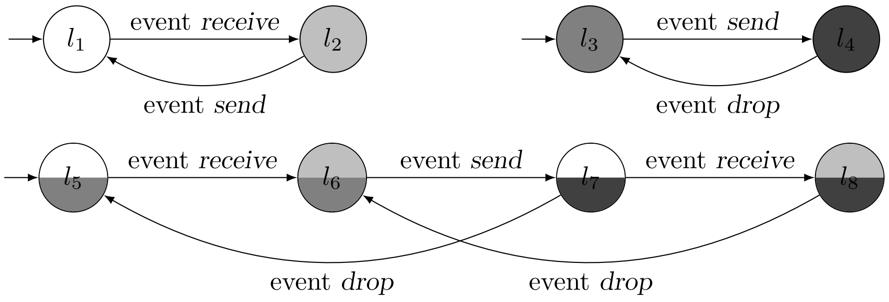


Figure 4.1: Synchronous product of two automata and the relation between their locations.

requirements, and/or features, are no longer traceable; clearly, when no colors are provided, the individual automata can no longer be distinguished in the synchronous product.

Traceability, or "*the ability to describe and follow the life of a requirement from its origins through any stage in the development*" (Gotel and Finkelstein, 1994; Winkler and von Pilgrim, 2010), is a topic in literature that is mostly used to assess the quality of software or models. Allowing for good traceability increases adequacy and understandability, while neglecting traceability can lead to less maintainability or defects.

Two examples are provided in the coming two paragraphs that show difficulties that arise with both the mentioned duplication of locations and transitions, as well as when traceability is neglected.

For verification, a simple query on the reachability of state l_4 in the original model changes in a query regarding the states l_7 and l_8 in the transformed model. One can imagine that it becomes virtually impossible to correctly, i.e., error-free, translate these queries manually when the state duplication worsens. Either the queries must be transformed together with the model, or the linkage between entities of the two models must be defined properly. Problems also arise if verification fails and a specific state is part of the returned counterexample. It may be very labour intensive to derive which component of the original model is responsible for the erroneous behavior. Again, this is more straightforward if the entities between the two models are properly linked.

For code generation, like verification, if erroneous behavior is observed during tests, it may prove very difficult and maybe even impossible to determine which component is responsible. However, another problem altogether arises when considering code generation. Namely, the duplication of states and transitions leads to duplication of code; the code that is created for a single transition, say the code associated with the transition labeled *receive*, is present multiple times at different locations throughout the code. This duplication will impair the comprehensibility and the adaptability of the code: it is no longer trivial to oversee all implications of a (small) change in the code. The adaptability remains of importance in many fields of industry where commissioning engineers have to adapt the software directly onsite.

The examples provided in the above paragraphs can be extended to any model-to-model transformation. For any transformed model that is used to provide some kind of feedback, a relation is needed to understand the results in terms of the original model.

It is trivial to see that the components remain traceable if exactly such a colored relation as provided in Figure 4.1 exists: using these colors it is exactly known which states and which transitions, i.e., the entities of the model, are linked together. Such a relation is called a traceability relation and this relation will relate the syntactical elements of two models.

In the literature, traceability relations for models are mostly contained in a metamodel (Aizenbud-Reshef et al., 2006) or within the model itself (Winkler and von Pilgrim, 2010), though a multitude of other solutions may be thought of. Most of the works referred to by Winkler and von Pilgrim (2010) focus on the textual linkage between written-down requirements and a model. In our modeling philosophy, each requirement is modeled separately and such a linkage is obtained almost free-of-charge.

In this chapter, the focus is on endogenous model-to-model transformations (Mens and Van Gorp, 2006); model-to-model transformations within the same modeling formalism. The initial linkage between requirement documents and the model, e.g., IBM Rational DOORS, and the final linkage between models of different formalisms or text, e.g., code generation, is not considered. In this case, an elaborate linkage as discussed in the above papers is not needed: when a specific requirement is linked to a specific automaton, it is sufficient to know how that specific automaton is transformed, i.e., which edges of the transformed model are related to the edges of the former requirement, to relink the requirement. If the model is known to the end-user, intuitively it is enough to provide a syntactical linkage between two models. In this case, the transformed model is understood by linking to entities of the understood original model (Woods and Yang, 1996).

A formal definition for a traceability relation will be provided in this chapter for automata models undergoing endogenous model-to-model transformations. This traceability relation is purely based on the syntactical elements of the model. Moreover, it is proven that these kinds of relations are transitive, so that multiple transformations can be chained while still being able to trace each entity.

In addition to traceability or a link between the syntactical elements of the model, it proves to be very important to assess the equivalence of the execution of two models or, in other words, the semantical linkage. For verification, for instance, it is key that each transition in the original model is mimicked in the transformed model and vice versa. Otherwise, erroneous behavior may be missed or added to the system. Executional equivalence is proven by finding a, so-called, bisimulation relation between the two models (Mousavi et al., 2005). As equivalence of execution is an important property for many model-to-model transformations, the formal definition of bisimilarity of models is also provided in this chapter.

In conclusion, in this chapter two relations are introduced; one between the syntax and one between the semantics of two models. Where the former is needed to relate the syntactical entities of the two models to allow traceability, the latter is needed to prove equivalence of execution of the two models.

4.2 Framework

In this chapter a formal syntax and semantics is used, resembling the syntax and the semantics of CIF as shown in Chapter 2. The major difference is the lack of algebraic equations and the concept of time. This is done to allow for straightforward proofs of the syntactical and semantical equivalence without the need of introducing new concepts to allow for equivalence with respect to time.

The syntax and semantics used in this chapter are introduced in this section. This may lead to some concepts being explained for a second time.

4.2.1 Automaton

Syntax As stated in the introduction, an automaton is a state machine and is commonly used to model controllers. An example of such an automaton is given in Figure 4.2, which models a specification for receiving and sending products.

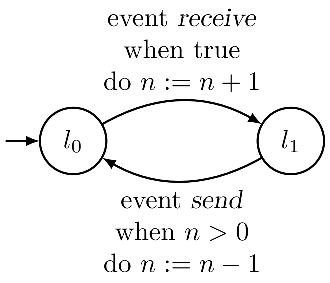


Figure 4.2: An automaton, modeling a specification for receiving and sending products.

The automaton consists of two *locations*, named l_0 and l_1. Locations model the discrete modes of the system. The actions of the system are defined by its *events*, i.e., *receive* and *send*. The set of the events of an automaton is called the alphabet. Additionally, data is modeled by variables, i.e., n. In this framework, variables are global entities and not part of the automaton itself.

The arrows between the locations are called *edges*, which model the transitions of the system. The edges are labeled with an event of the alphabet. Moreover, the condition under which the transition can occur is denoted on the edge after the keyword “when”. This condition is called the *guard*. Finally, an edge contains an *update* after the keyword “do”. All updates are modeled as assignments, i.e., $x := e$, where $x \in \boldsymbol{X}$ is a variable and $e \in E(\boldsymbol{X})$ an expression over the variables of $\boldsymbol{X}$.

A transition can only occur if the source location of the edge modeling this transition is active. The active location is modeled by a small incoming arrow. In Figure 4.2, location l_0 is active.

Formally, an automaton over the universe of variables $\boldsymbol{X}$ is defined by a quadruple (4-tuple):

$$(L, \Sigma, \Delta, l)$$

Here, $L \subseteq \boldsymbol{L}$ is the set of locations, $\Sigma \subseteq \boldsymbol{\Sigma}$ is the set of events (or alphabet), $P(\boldsymbol{X})$ is the set of predicates over the variables, $U(\boldsymbol{X})$ is the set of updates, $\Delta \subseteq L \times P(\boldsymbol{X}) \times \Sigma \times U(\boldsymbol{X}) \times L$ is the set of edges, and $l \in L$ is the active location. The universe of automata is denoted by $\boldsymbol{A}$.

Semantics The state of an automaton is defined by the active location and the valuation, i.e., a mapping from variables to values. The execution of an edge, i.e., a transition, induces a change of active location and a change of variable values. As an example, consider Figure 4.3.

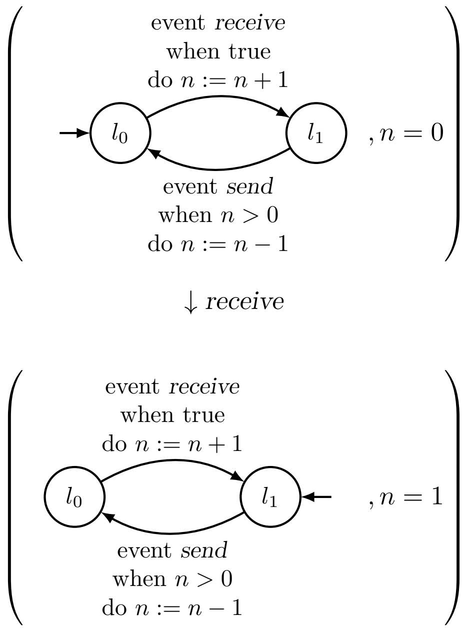


Figure 4.3: The semantics of the transition labeled with the event *receive*.

The left-hand side of Figure 4.3, between the parentheses, denotes the initial state. It is assumed that an initial valuation is provided. In this particular case, a value of zero for n is assumed. Note that the automaton as a whole is present in the state. Providing the automaton as part of the state simplifies the semantics of compositions, as introduced later.

A transition can occur, if the source location of the edge is active and the guard is satisfied. Considering Figure 4.3, the transition labeled *receive* can occur, as the source location is active and the guard is satisfied with respect to the valuation. The transition induces a change of the value of n according to the assignment and a change of the active location to the target location. After the transition, the situation is obtained that is depicted in the right-hand side.

The target state depicted in Figure 4.3 can be considered as a source state for another transition. It is not difficult to see that the transition with the event *send* can occur from this state. Note that in this case the actual value of n is of importance, as the guard specifies a condition on this value. After the transition, the state depicted in the left-hand side of Figure 4.3 is obtained again.

Formally, the semantics of an automaton can be defined in a transition system. A transition system contains the state of the system and the transitions between these states, defined as:

$$(S, \longrightarrow, s_0)$$

With $S \subseteq \boldsymbol{A} \times (\boldsymbol{X} \to \boldsymbol{V})$ the set of states, $\longrightarrow \subseteq S \times \boldsymbol{\Sigma} \times S$ the set of transitions, $s_0 \in S$ the initial state of the transition system, and $\boldsymbol{V}$ the universe of proper values of the variables.

The set of transitions is defined by means of Structural Operational Semantics (SOS) (Mousavi et al., 2005). The rule for a transition is given as follows:

$$\frac{(l_s, g, \sigma, u, l_t) \in \Delta, v \models g, v \cup v' \models u}{((L, \Sigma, \Delta, l_s), v) \xrightarrow{\sigma} ((L, \Sigma, \Delta, l_t), v')}$$

Herein, $v \models g$ is used to denote that the guard g is satisfied with the values defined by the valuation v. $v \cup v' \models u$ is used to denote that the current valuation v and the future valuation v' together must satisfy the update u. As updates are given by assignments, e.g., $x := y$, this implies that x is bound in v' based on the current value of y, defined by v.

Note that in this framework all variables not mentioned in the update may change value freely. If the value of a variable x must remain equal, an explicit assignment $x := x$ must be added to the update.

Remark In this and the following chapter, it is assumed that all updates are non-conflicting. Two updates are non-conflicting if both updates agree on all assigned variable values. Under this assumption, the semantical rules can be simplified.

Additionally, it is assumed that no redundant edges are present in automata. A redundant edge is an edge that does not add more behavior to the automaton. The reason is due to the nature of the traceability relation, which will become clear when the relations are introduced. An example of a redundant edge is given in Figure 4.4. The gray edge is clearly redundant, because the update u is executed when guard g is satisfied, regardless of whether or not guard g' is satisfied.

4.2.2 Composition

Syntax To allow the creation of a larger model from multiple automata, composition is introduced. The parallel composition operator, i.e., $\|$, is used to denote the parallel execution of automata, synchronizing on equally labeled events. Formally, the grammar of compositions is defined as:

$$\begin{array}{rcll} \boldsymbol{C} ::= & \boldsymbol{A} & & \text{automaton} \\ | & \boldsymbol{C} \parallel \boldsymbol{C} & & \text{parallel composition} \end{array}$$

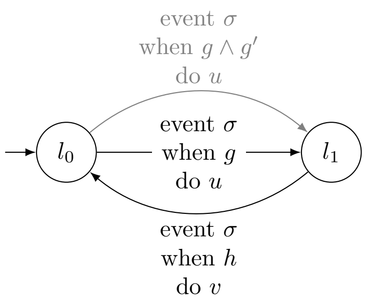


Figure 4.4: A redundant edge.

Semantics For the semantics of a composition of multiple automata, it is of importance whether synchronization can take place; the transition associated with a shared event can only occur in the composition when in both automata a transition labeled with this particular event is possible.

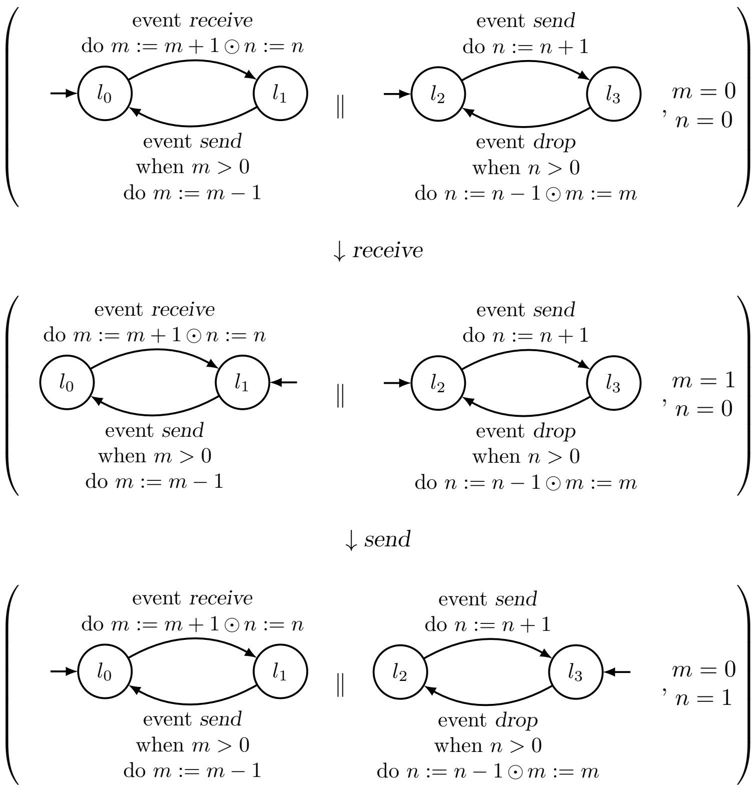


Figure 4.5: The semantics of a synchronizing transition in a composition.

Consider the system of Figure 4.5, with the given initial valuation. In the composition, event *send* is shared and, as a result, needs to be synchronized: only when in both automata a transition labeled with event *send* is possible, the transition is executed.

Event *send* is currently not possible in the leftmost automaton: in the initial state, only event *receive* is possible. Namely, the source location is active and the guard is satisfied. As this event is not shared, the leftmost automaton will change state according to the semantics of a single automaton. Do note that, in order to prevent the variable n from randomly jumping value, a self-assignment is explicitly defined in the updates. This is only done for the sake of simplicity, as otherwise multiple valuations would have been possible.

After the execution of *receive*, event *send* becomes possible in both automata: the guards are satisfied and the source locations are active. By executing the transition, both automata change state synchronously. This is depicted in the lower part of Figure 4.5.

The set of (deduction) rules is naturally extended with those for compositions. Note that this implies that the state is a composition, i.e., $S \subseteq \boldsymbol{C} \times (\boldsymbol{X} \to \boldsymbol{V})$. The notation $\Sigma(c)$ is used to denote the alphabet of the composition c. For a single automaton it is the alphabet of the automaton. Otherwise, $\Sigma(c_1 \parallel c_2) = \Sigma(c_1) \cup \Sigma(c_2)$.

First, the synchronous execution of a transition is defined:

$$\frac{(c_1, v) \xrightarrow{\sigma} (c_1', v'), (c_2, v) \xrightarrow{\sigma} (c_2', v'), \sigma \in \Sigma(c_1) \cap \Sigma(c_2)}{(c_1 \parallel c_2, v) \xrightarrow{\sigma} (c_1' \parallel c_2', v')}$$

Finally, the non-synchronous transitions are defined:

$$\frac{(c_1, v) \xrightarrow{\sigma} (c_1', v'), \sigma \in \Sigma(c_1) \setminus \Sigma(c_2)}{(c_1 \parallel c_2, v) \xrightarrow{\sigma} (c_1' \parallel c_2, v')} \qquad \frac{(c_2, v) \xrightarrow{\sigma} (c_2', v'), \sigma \in \Sigma(c_2) \setminus \Sigma(c_1)}{(c_1 \parallel c_2, v) \xrightarrow{\sigma} (c_1 \parallel c_2', v')}$$

4.3 Traceability relation

For two models to be (syntactically) related, there must be a (traceability) relation between the locations, alphabets, and edges of the automata. As syntactical changes can occur in automata due to (model-to-model) transformations, the locations, the event labels, and/or the edges of the automata may be altered. In the traceability relation these changes must be allowed.

Two sets of automata are traceable if a relation can be found between the edges of these automata. Each edge of each automaton in the first set must relate to an edge of an automaton in the second set, and vice versa. As an example, consider Figure 4.6 where the concept of traceability is schematically depicted. The relation between the edges is denoted by ψ_{Δ}.

Zooming in, two edges are related when one edge is more restrictive than the other. Note that a direction is present here in the linkage, from less to more restrictive. When an edge is more restrictive than another, the guard of the more restrictive edge is a strengthened variant of the guard of the less restrictive edge. In this case, at least the original conditions (or the original requirement(s)) are met. For the update, the

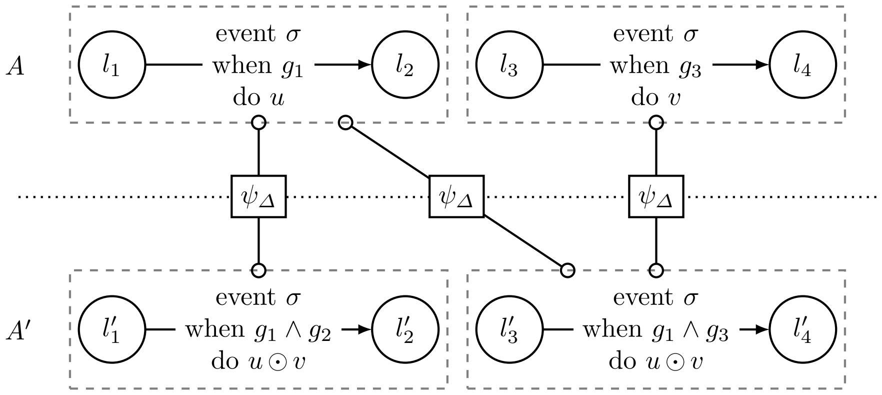


Figure 4.6: Schematic overview of the traceability relation between A and A'.

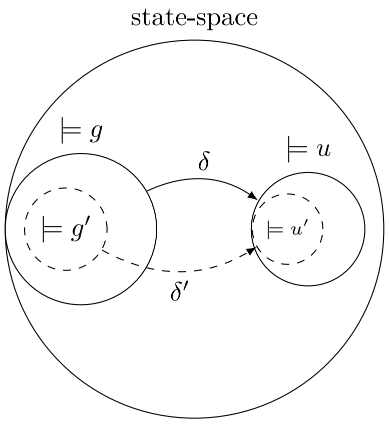


Figure 4.7: Schematic representation of restrictiveness in the state-space of an automaton.

more restrictive edge should assign more variables but, at least, the same values to the same variables.

Graphically the restrictiveness can be captured by Figure 4.7 where the difference between the state-spaces is drawn between the less and more restrictive edge. The outer circle denotes the complete state-space of the system. Taking a guard g and update u associated with transition δ, the leftmost solid circle denotes the state-space for which this guard is satisfied. The rightmost solid circle denotes the state-space which satisfies the update, where the transition δ links the two spaces. A similar relation is shown between the more restrictive guard $g' \implies g$ and the more restrictive update $u' \geq u$ and their corresponding state-spaces (denoted by the dashed circles).

Given a relation between the locations and the event labels, Figure 4.8 depicts the relation between the edges. The bottommost edge is a more restrictive version of the topmost edge. As stated, the bottommost edge contains part of the dynamics of the

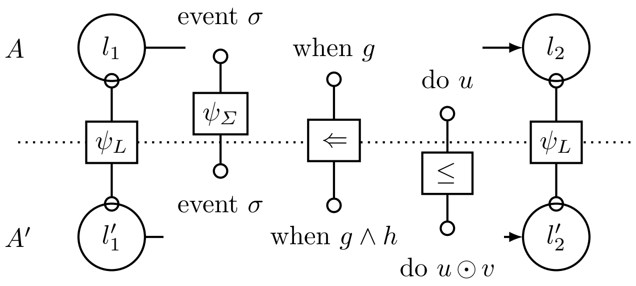


Figure 4.8: Schematic overview of the relation between an edge and a more restrictive edge.

topmost edge; the guard of the bottommost edge implies the guard of the topmost edge, and the update of the bottommost edge assigns more variables than the update of the topmost edge.

Definition 9 (Witness). *Given a transitive relation $\psi_L \subseteq \boldsymbol{L} \times \boldsymbol{L}$ and a transitive relation $\psi_\Sigma \subseteq \boldsymbol{\Sigma} \times \boldsymbol{\Sigma}$, $\psi_\Delta(\psi_L, \psi_\Sigma) \subseteq \boldsymbol{\Delta} \times \boldsymbol{\Delta}$ is a witness for traceability, iff*

$$\forall((l_s, g, \sigma, u, l_t), (l'_s, g', \sigma', u', l'_t)) \in \psi_\Delta(\psi_L, \psi_\Sigma) : \begin{cases} (l_s, l'_s) & \in & \psi_L \\ (l_t, l'_t) & \in & \psi_L \\ (\sigma, \sigma') & \in & \psi_\Sigma \\ g & \Leftarrow & g' \\ u & \leq & u' \end{cases}$$

Where $u \leq u'$ denotes that u is less or equally restrictive than u', defined as

$$u \leq u' \iff \forall v, v' \in \boldsymbol{X} \to \boldsymbol{V} : (v \cup v' \models u \Leftarrow v \cup v' \models u')$$

Formally, the definition of traceability is given, as follows:

Definition 10 (Traceability). *Given two sets of automata, $A, A' \subseteq \boldsymbol{A}$, a transitive relation $\psi_L \subseteq \boldsymbol{L} \times \boldsymbol{L}$, a transitive relation $\psi_\Sigma \subseteq \boldsymbol{\Sigma} \times \boldsymbol{\Sigma}$, and a witness $\psi_\Delta(\psi_L, \psi_\Sigma) \subseteq \boldsymbol{\Delta} \times \boldsymbol{\Delta}$, the set A is traceable with respect to the set A', and vice versa, denoted by $A \leftrightarrows_{\psi_\Delta(\psi_L, \psi_\Sigma)} A'$, iff*

$$\forall (L, \Sigma, \Delta, l) \in A, \forall \delta \in \Delta, \exists (L', \Sigma', \Delta', l') \in A', \exists \delta' \in \Delta' : (\delta, \delta') \in \psi_\Delta(\psi_L, \psi_\Sigma)$$
$$\forall (L', \Sigma', \Delta', l') \in A', \forall \delta' \in \Delta', \exists (L, \Sigma, \Delta, l) \in A, \exists \delta \in \Delta : (\delta, \delta') \in \psi_\Delta(\psi_L, \psi_\Sigma)$$

Commonly, successive transformations are applied to the model, one after another. Therefore, it is important that witnesses are transitive:

Lemma 4 (Transitivity). *Given three sets of automata A, A', A'', a transitive relation $\psi_L \subseteq \boldsymbol{L} \times \boldsymbol{L}$, a transitive relation $\psi_\Sigma \subseteq \boldsymbol{\Sigma} \times \boldsymbol{\Sigma}$, and two witnesses*

$$\psi_\Delta(\psi_L, \psi_\Sigma), \psi'_\Delta(\psi_L, \psi_\Sigma) \subseteq \boldsymbol{\Delta} \times \boldsymbol{\Delta}$$

such that $A \leftrightarrows_{\psi_{\Delta}(\psi_L,\psi_\Sigma)} A'$ *and* $A' \leftrightarrows_{\psi'_{\Delta}(\psi_L,\psi_\Sigma)} A''$*, then* $A \leftrightarrows_{\psi''_{\Delta}(\psi_L,\psi_\Sigma)} A''$ *where*

$$\psi''_{\Delta}(\psi_L,\psi_\Sigma) = \{(\delta,\delta'') \mid \exists \delta' \in \boldsymbol{\Delta}, (\delta,\delta') \in \psi_{\Delta}(\psi_L,\psi_\Sigma), (\delta',\delta'') \in \psi'_{\Delta}(\psi_L,\psi_\Sigma)\}$$

.

Proof. First, it must be proven that $\psi''_{\Delta}(\psi_L,\psi_\Sigma)$ is a witness. Given an element

$$((l_s,g,\sigma,u,l_t),(l'_s,g',\sigma',u',l'_t)) \in \psi_{\Delta}(\psi_L,\psi_\Sigma)$$

and an element

$$((l'_s,g',\sigma',u',l'_t),(l''_s,g'',\sigma'',u'',l''_t)) \in \psi'_{\Delta}(\psi_L,\psi_\Sigma)$$

it holds that

$$\begin{aligned}
(l_s,l'_s) \in \psi_L \wedge (l'_s,l''_s) \in \psi_L &\implies (l_s,l''_s) \in \psi_L \\
(l_t,l'_t) \in \psi_L \wedge (l'_t,l''_t) \in \psi_L &\implies (l_t,l''_t) \in \psi_L \\
(\sigma,\sigma') \in \psi_\Sigma \wedge (\sigma',\sigma'') \in \psi_\Sigma &\implies (\sigma,\sigma'') \in \psi_\Sigma \\
g \Leftarrow g' \wedge g' \Leftarrow g'' &\implies g \Leftarrow g'' \\
(v \models u' \Leftarrow v \models u'') \wedge (v \models u \Leftarrow v \models u') &\implies v \models u \Leftarrow v \models u''
\end{aligned}$$

Hence, $\psi''_{\Delta}(\psi_L,\psi_\Sigma)$ is a witness.

Finally, it is proven that $\psi''_{\Delta}(\psi_L,\psi_\Sigma)$ is a witness for $A \leftrightarrows_{\psi''_{\Delta}(\psi_L,\psi_\Sigma)} A''$. From the definition of traceability it is known that

$$\begin{gathered}
\forall a \in A, \forall \delta \in \Delta, \exists a' \in A', \exists \delta' \in \Delta' : (\delta,\delta') \in \psi_{\Delta}(\psi_L,\psi_\Sigma) \\
\forall a' \in A', \forall \delta' \in \Delta', \exists a \in A, \exists \delta \in \Delta : (\delta,\delta') \in \psi_{\Delta}(\psi_L,\psi_\Sigma) \\
\forall a' \in A', \forall \delta' \in \Delta', \exists a'' \in A'', \exists \delta'' \in \Delta'' : (\delta',\delta'') \in \psi'_{\Delta}(\psi_L,\psi_\Sigma) \\
\forall a'' \in A'', \forall \delta'' \in \Delta'', \exists a' \in A', \exists \delta' \in \Delta' : (\delta',\delta'') \in \psi'_{\Delta}(\psi_L,\psi_\Sigma)
\end{gathered}$$

which implies

$$\begin{gathered}
\forall a \in A, \forall \delta \in \Delta, \exists a'' \in A'', \exists \delta'' \in \Delta'' : (\delta,\delta') \in \psi_{\Delta}(\psi_L,\psi_\Sigma) \wedge (\delta',\delta'') \in \psi'_{\Delta}(\psi_L,\psi_\Sigma) \\
\forall a'' \in A'', \forall \delta'' \in \Delta'', \exists a \in A, \exists \delta \in \Delta : (\delta',\delta'') \in \psi'_{\Delta}(\psi_L,\psi_\Sigma) \wedge (\delta,\delta') \in \psi_{\Delta}(\psi_L,\psi_\Sigma)
\end{gathered}$$

Hence, $A \leftrightarrows_{\psi''_{\Delta}(\psi_L,\psi_\Sigma)} A''$. □

4.4 Bisimulation relation

When transforming one model to another, it may be of importance that the executional behavior of the model remains equal[3]. Equality of executional behavior is of importance for transformations related to model verification, for example. To make sure all erroneous behavior is found, the transformed model must contain the same executional behavior.

[3]Note that there may be transformations for which it is sufficient that part of the executional behavior remains.

To compare two transition systems, a notion of equivalence is introduced. Two transition systems are equivalent if a relation can be found that relates the states of the two transition systems and the transitions between states.

In this chapter, a bisimulation relation is defined under partial observation of variables. This entails that not all variables and associated valuations are considered when finding a bisimulation relation. This is especially helpful when new variables are introduced during transformations that where not present in the original model. Bisimulation under partial observation only relates the values of a predefined set of variables. In such a way, the newly introduced variables can be removed from observation. Bisimulation under partial observation can be categorized between (standard) bisimulation (Cassandras and Lafortune, 2009), and stateless bisimulation (Mousavi et al., 2005).

Definition 11 (Bisimulation under partial observation). *Let $t_1 = (S_1, \longrightarrow_1, s_{0,1})$ and $t_2 = (S_2, \longrightarrow_2, s_{0,2})$ be two transition systems and let $X \subseteq \boldsymbol{X}$ be the set of observable variables. Remember that $S \subseteq \boldsymbol{C} \times (\boldsymbol{X} \to \boldsymbol{V})$. The transition systems t_1 and t_2 are bisimilar under observation of $X \subseteq \boldsymbol{X}$, denoted by $t_1 \overset{X}{\sim} t_2$, if there exists a bisimulation relation $R(X)$ such that the initial states are related, i.e., $(s_{0,1}, s_{0,2}) \in R(X)$. A binary relation $R(X) \subseteq S_1 \times S_2$ is a bisimulation if, and only if, for all pairs $(s_1, s_2) \in R(x)$ with $s_i = (c_i, v_i)$ it holds that*

$$\forall x \in X : v_1(x) = v_2(x)$$

and

$$\forall \sigma \in \boldsymbol{\Sigma} : \begin{cases} \forall s_1' \in S_1 : s_1 \xrightarrow{\sigma}_1 s_1' \implies \exists s_2' \in S_2 : s_2 \xrightarrow{\sigma}_2 s_2' \wedge (s_1', s_2') \in R(X) \\ \forall s_2' \in S_2 : s_2 \xrightarrow{\sigma}_2 s_2' \implies \exists s_1' \in S_1 : s_1 \xrightarrow{\sigma}_1 s_1' \wedge (s_1', s_2') \in R(X) \end{cases}$$

Transitivity for the equivalence of executional behavior, i.e., transitivity of the bisimulation relation, is needed.

Lemma 5 (Transitivity bisimulation). *Given three compositions, $c_1, c_2, c_3 \in \boldsymbol{C}$ and two bisimulation relations $R(X), R'(X)$ proving, respectively, $c_1 \overset{X}{\sim} c_2$ and $c_2 \overset{X}{\sim} c_3$, then $R''(X) = R'(X) \circ R(X)$ is a bisimulation relation proving $c_1 \overset{X}{\sim} c_3$.*

Proof. Suppose, $(x, z) \in R''(X)$ and $x \xrightarrow{\sigma} x'$. There must be a state y such that $(x, y) \in R(X)$ and a transition $y \xrightarrow{\sigma} y'$ such that $(x', y') \in R(X)$. Likewise, via $R'(X)$, a relation is made between y and z and y' and z'. This implies that for any transition $x \xrightarrow{\sigma} x'$ there is a transition $z \xrightarrow{\sigma} z'$ with, clearly, $(x', z') \in R''(X)$. □

4.5 Concluding remarks

Note that two automata models that are semantically equivalent (bisimilar) do not necessary need to be syntactically relatable. There may exist numerous models that have the same executional behavior, but are syntactically not relatable. The other way around, two models that are traceable (or syntactically related), do not necessary need to be semantically equivalent.

Chapter 5

Removal of synchronous behavior

This chapter focusses on the removal of synchronization. It is shown that for code generation it might be better to introduce an alternative algorithm to remove synchronization. An specialized algorithm is introduced for which it is proven that the resulting model is traceable and similar to the original model.

5.1 Introduction

As discussed, the removal of synchronization is necessary to allow the usage of models (containing synchronization) in other formalisms that do not support synchronization. The goal is to provide a transformation that removes the synchronization such that another transformation can do the platform-specific transformation. A visual depiction of this approach is given in Figure 5.1 where the common part, i.e., the removal of synchronization, is captured by a single transformation: the individual transformations, i.e., t_1 through t_3, are replaced by a single transformation that removes synchronization, i.e., t, and platform-specific transformations, i.e., t'_1 through t'_3.

The question arises why this approach may be preferred for code generation. Namely, an alternative to computing the synchronous product is shown by Park and Kim (1990) where an extra control layer is added to the controller code that handles synchronization. A control process is either a slave or a master. Each slave must show its *willingness* to synchronize. Each master waits for all other processes until their *willingness* is expressed. If all parties agree, the synchronous communication is handled.

Although the mentioned protocol allows the synchronization of multiple parties (automata), communication between the processes is needed to reach a consensus on synchronization. This leads to additional overhead, both in the addition of extra code as well in execution time. Additionally, the impact of this protocol on the traceability is not straightforward, as a shift is made from (synchronizing) events to messages.

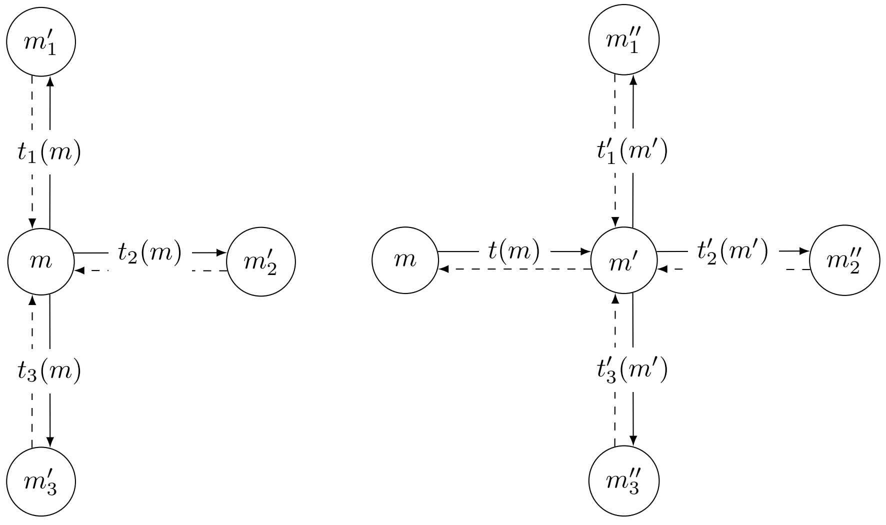


Figure 5.1: The relations between multiple models. The model-to-model transformations are depicted as solid lines, while the traceability relations are related by the dotted lines.

PLC specific approaches are given by Heegren et al. (1999); Cengic et al. (2005). In these papers, the model is converted into several objects that are executed independently until an event needs to be synchronized. This is achieved using *Sequential Function Charts* (SFCs) by Heegren et al. (1999) and *Function Blocks* (FBs) by Cengic et al. (2005). It must be noted that the latter paper uses a newer PLC standard, i.e., IEC 61499, while the former paper uses the more commonly used IEC 61131.

The proposed methods can be most likely extended to other control platforms, but these methods will remain implementation-specific solutions. As a result, each model-to-model transformation may obtain a completely different traceability relation; code that is generated for a PLC platform may be structurally completely different than code generated for a C or Java platform.

All of these changes induce difficulties for commissioning engineers which frequently have to make small modifications to the controller code on-site. These tweaks are mostly related to differences in dynamics, i.e., the differences between the ideal (modeling) world and the actual system. In order to allow these tweaks, the generated code must be traceable but must also be adaptable. Intuitively, code is adaptable if a change in code is easily implemented and the result of the change can be foreseen[4]. Note that adaptability can be interpreted in many ways and depends on the wishes of the end-user. A formal definition of adaptability and proofs on the adaptability of results models is therefore omitted in this chapter. Nevertheless, intuitively the easiness of adapting code may be connected to the traceability; using a traceability relation it is clear which parts of the code need to be altered when specific requirements (and

[4]Up to a certain degree, as no change in the code can be completely foreseen when a complex system is considered.

associated transitions) are adapted.

When considering the removal of synchronous events, multiple transformations can be thought of that do not impair model traceability and for which the transformed model is in bisimulation with the original model. However, additional properties are also of importance when verification and code generation are considered, as highlighted earlier. An alternative algorithm for the removal of synchronization is provided for which the resulting model is traceable and in bisimulation with the original model, and which does not duplicate non-shared events as could be observed in Chapter 4.

To provide some insights on the course of this chapter, consider Figure 5.2 and assume the top two automata are part of a model of a small system. It has been shown in Figure 4.1 that the synchronous product induces location and transition duplication. Ideally, a model as shown by the bottom two automata would be preferred as there is no location or transition duplication. Synchronization has been removed by computing the synchronous behavior for the transitions labeled *send*, using the semantics of synchronizing events as provided in Chapter 4. The complete dynamics has been placed on, at the moment arbitrarily, the leftmost of the two edges while the other edge is removed. Note that the removal of the edge impairs the actual execution of this model: there is no longer a transition possible from l_3 to l_4.

If the introduced (semantical) problem of changing locations can be solved, the "synchronous product" depicted at the bottom of Figure 5.2 would be beneficial for code generation as it will prevent code duplication (by preventing duplication of transitions). In this chapter, a solution is provided to this problem by means of model transformations. For each of the introduced transformations, full traceability is obtained between the transformed and original model. Moreover, it is proven that each transformed model is in bisimulation with the original model. Note that these transformations will have a large visual impact on the automata, while the actual syntactical changes are less severe. As the syntactical changes are less severe, the solution suffices for code generation which depends largely on the syntax of the automata.

5.2 Implications

Before the details of the transformations are provided in the following sections, this section provides a very abstract overview on the implications of moving the synchronous dynamics to a specific edge. In Figure 5.2, the shared event is present only once in each automaton, making the choice where to move the resulting dynamics arbitrary.

Now, consider Figure 5.3 where the shared event is present twice in the rightmost automaton. Still using the intuitive, but semantically incorrect, removal of synchronization, multiple solutions can be created that remove the synchronous behavior. Two possible solutions are provided in this section with accompanied traceability relations.

Looking simultaneously to Figure 5.4 and Figure 5.5, it is clear that the choice of moving the synchronous dynamics to the topmost automaton or the bottommost automaton induces a similar traceability relation; the only difference between the traceability relation is due to the difference in source and target locations. Neverthe-

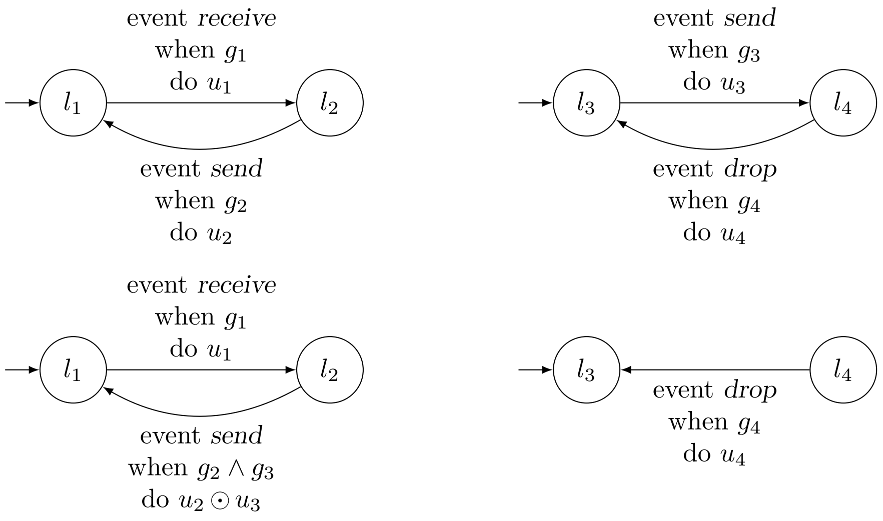


Figure 5.2: Overview of the removal of synchronous events; top part shows the original model while the bottom part shows the desired result.

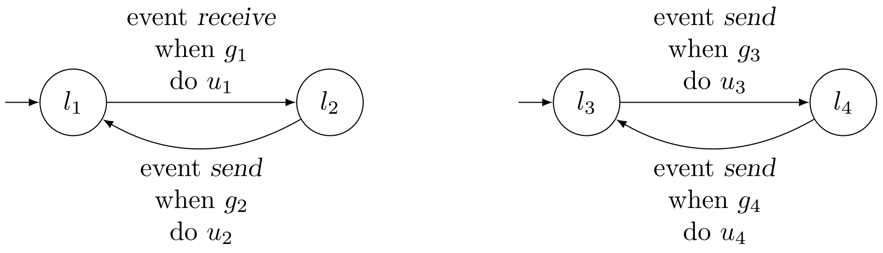


Figure 5.3: Model which contains multiple equally labeled events in the rightmost automaton.

less, there is a slight structural difference between the two models[5]. Different needs on the structure of the transformed model may result in a preference of moving the dynamics to either the topmost or bottom automaton. As the choice is arbitrary for code generation, the choice is left to the end-user; a preference of automaton for each event may be posed during the transformation.

[5]This structural difference will even reduce further when superposition of locations has been introduced.

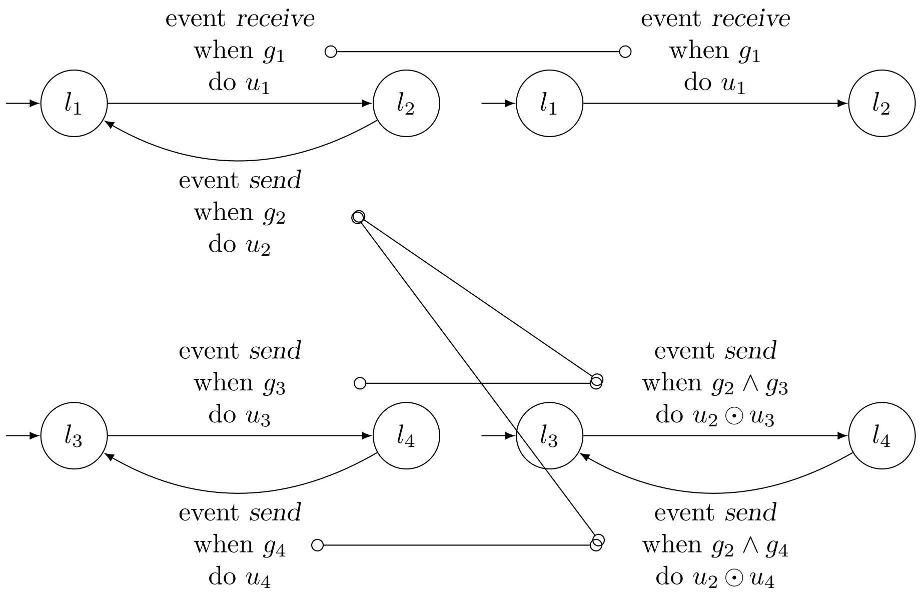


Figure 5.4: Moving the dynamics to the bottommost automaton.

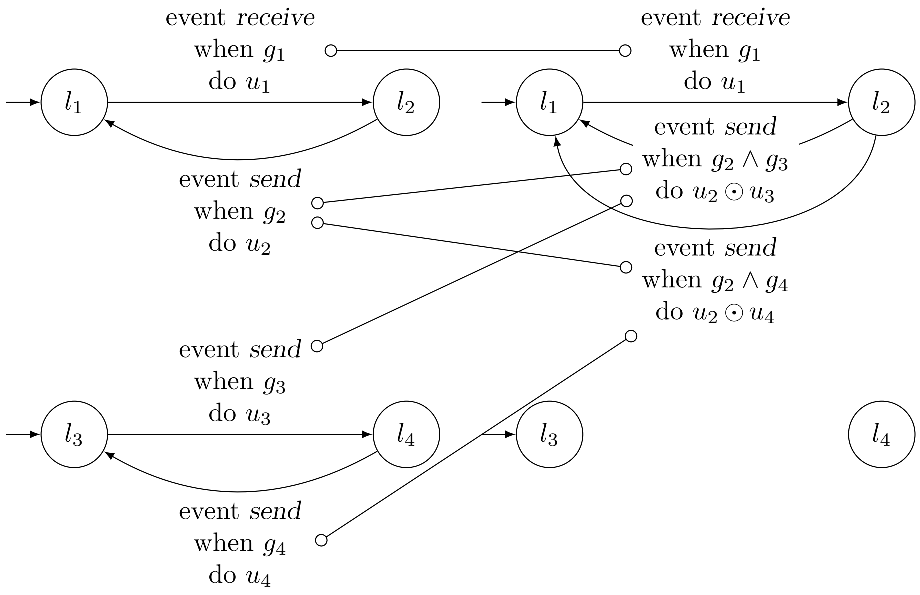


Figure 5.5: Moving the dynamics to the topmost automaton.

5.3 Informal

5.3.1 Location vocalization

A non-structural model-to-model transformation is the "location vocalization" transformation shown in Figure 5.6. This transformation transform the location information to data, i.e., vocalizes the locations by adding the locations explicitly to the guards and assignments. This transformation will prove to be key in the reduction of transition and location duplication during the removal of synchronous behavior. To vocalize the locations, a so-called "location pointer" is introduced and a vocalization function $\omega : \boldsymbol{L} \to \boldsymbol{V}$ which transforms a location into the domain of values.

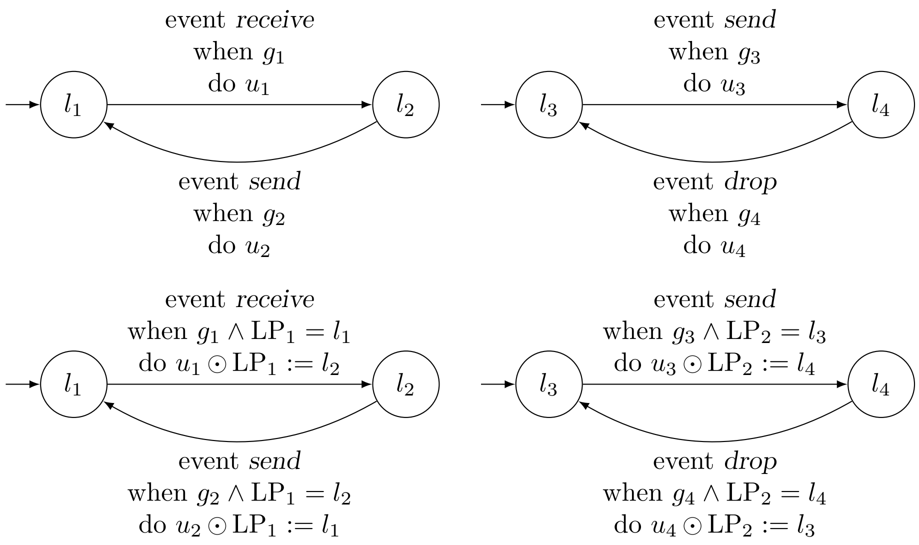
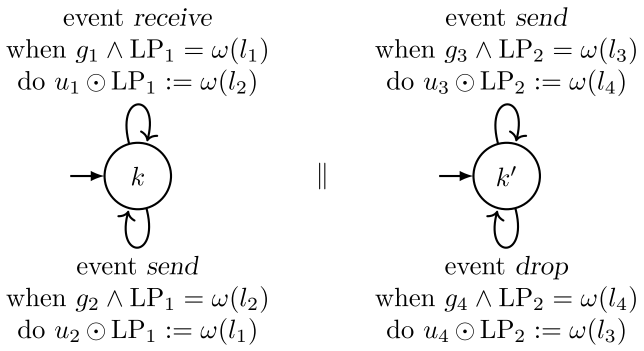

event *receive*
when g_1
do u_1
l_1
l_2
event *send*
when g_2
do u_2
event *send*
when g_3
do u_3
l_3
l_4
event *drop*
when g_4
do u_4
event *receive*
when $g_1 \wedge \mathrm{LP}_1 = l_1$
do $u_1 \odot \mathrm{LP}_1 := l_2$
l_1
l_2
event *send*
when $g_2 \wedge \mathrm{LP}_1 = l_2$
do $u_2 \odot \mathrm{LP}_1 := l_1$
event *send*
when $g_3 \wedge \mathrm{LP}_2 = l_3$
do $u_3 \odot \mathrm{LP}_2 := l_4$
l_3
l_4
event *drop*
when $g_4 \wedge \mathrm{LP}_2 = l_4$
do $u_4 \odot \mathrm{LP}_2 := l_3$

Figure 5.6: Vocalization of locations.

Note that vocalization also solves the problem of location duplication for verification when the synchronous product is computed. As we now have access to location pointers, queries over the locations can be easily rewritten as queries over the location pointers. In the synchronous product the location combining the former locations l_1 and l_3, for instance, vocalizes this by means of a valuation of $\mathrm{LP}_1 = \omega(l_1), \mathrm{LP}_2 = \omega(l_3)$; the actual locations as now present in the synchronous product are no longer of importance.

Also note that if the location pointers could be used to substitute the locations, the semantical problem of changing the location discussed in the introduction (Figure 5.2) can be solved. To this end, the next section introduces a transformation that removes all locations expect one from the automaton.

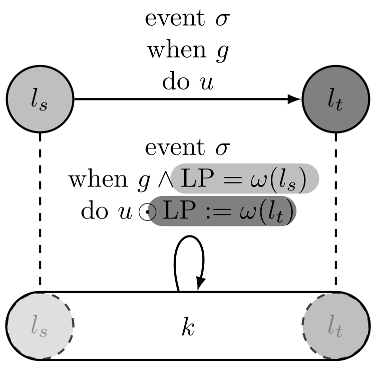


event *receive*
when $g_1 \wedge \mathrm{LP}_1 = \omega(l_1)$
do $u_1 \odot \mathrm{LP}_1 := \omega(l_2)$
k
event *send*
when $g_2 \wedge \mathrm{LP}_1 = \omega(l_2)$
do $u_2 \odot \mathrm{LP}_1 := \omega(l_1)$
$\parallel$
event *send*
when $g_3 \wedge \mathrm{LP}_2 = \omega(l_3)$
do $u_3 \odot \mathrm{LP}_2 := \omega(l_4)$
k'
event *drop*
when $g_4 \wedge \mathrm{LP}_2 = \omega(l_4)$
do $u_4 \odot \mathrm{LP}_2 := \omega(l_3)$

Figure 5.7: Superposition of locations; all locations combined to a single location.

5.3.2 Superposition of locations

To remove the locations expect one, all locations are superimposed to create a single (super) location. This is achieved by transforming each edge to a selfloop for which the source and target location is the single remaining location. As an example, Figure 5.7 depicts the situation after the removal of locations from the system of Figure 5.6.



event σ
when g
do u
l_s
l_t
event σ
when $g \wedge \mathrm{LP} = \omega(l_s)$
do $u \odot \mathrm{LP} := \omega(l_t)$
l_s
k
l_t

Figure 5.8: Schematic overview of the relations between LP and the (former) locations.

If a visual comparison is made between Figure 5.6 and Figure 5.7, the change in structure looks quite dramatic. The actual change, i.e., the change in the automaton tuple, is less grave. Moreover, the visual components can be easily related, as shown in Figure 5.8.

5.3.3 Superposition of dynamics

Finally, the dynamics of synchronizing events is superposed by moving the combined dynamics to one, arbitrarily chosen, edge as shown in Figure 5.9. Drawing a parallel with Figure 5.2 it is not difficult to see the resemblance; to overcome the semantically

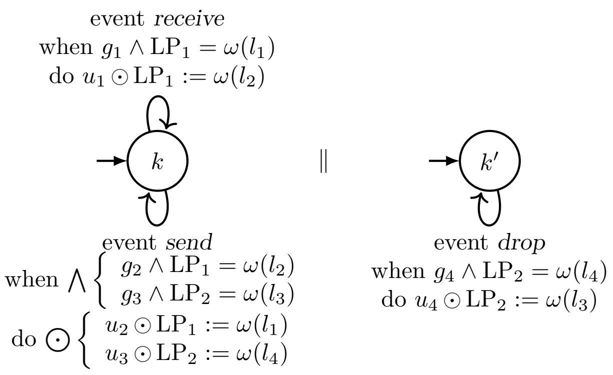

event *receive*
when $g_1 \wedge \mathrm{LP}_1 = \omega(l_1)$
do $u_1 \odot \mathrm{LP}_1 := \omega(l_2)$
k
$\|$
k'
event *send*
when $\bigwedge \begin{cases} g_2 \wedge \mathrm{LP}_1 = \omega(l_2) \\ g_3 \wedge \mathrm{LP}_2 = \omega(l_3) \end{cases}$
do $\bigodot \begin{cases} u_2 \odot \mathrm{LP}_1 := \omega(l_1) \\ u_3 \odot \mathrm{LP}_2 := \omega(l_4) \end{cases}$
event *drop*
when $g_4 \wedge \mathrm{LP}_2 = \omega(l_4)$
do $u_4 \odot \mathrm{LP}_2 := \omega(l_3)$

Figure 5.9: Superposition of the dynamics of synchronizing events.

wrongful deletion of the edge from the rightmost automaton, the location pointers and the selfloops have been introduced.

5.4 Formal

5.4.1 Location vocalization

Given a mapping $\omega : \boldsymbol{L} \to \boldsymbol{V}$, the location pointer LP is formally introduced in the guards and updates with the following transformation:

$$\mathrm{intro}((l_s, g, \sigma, u, l_t), \omega) = (l_s, g \wedge \mathrm{LP} = \omega(l_s), \sigma, u \odot \mathrm{LP} := \omega(l_t), l_t)$$

This function can be naturally extended to automata and models, such that:

$$\begin{array}{rcl} \mathrm{intro}((L, \Sigma, \Delta, l), \omega) & = & (L, \Sigma, \{\mathrm{intro}(\delta, \omega) \mid \delta \in \Delta\}, l) \\ \mathrm{intro}(A, \omega) & = & \{\mathrm{intro}(a, \omega) \mid a \in A\} \end{array}$$

Lemma 6. *Let* $\psi_L = \mathrm{id}$ *and* $\psi_\Sigma = \mathrm{id}$. *Given a set of automata* $A \subseteq \boldsymbol{A}$, *it holds that* $A \leftrightarrows \mathrm{intro}(A, \omega)$.

Proof. A witness is given by:

$$\psi_\Delta = \{(\delta, \mathrm{intro}(\delta, \omega)) \mid \delta \in \boldsymbol{\Delta}\}$$

Under the assumption that each edge is non-redundant, this is the largest witness with respect to A and A'. Namely, it is clear that no edges are introduced or removed: each edge is transformed to a more restrictive version and there is a strict one-to-one mapping. □

As the location pointer variables are not assigned in the original automaton, the value of these variables may jump freely in the original model. Therefore, bisimulation under observation of all variables except the location pointer is proven.

Lemma 7. *Given an automaton $a = (L, \Sigma, \Delta, l) \in \boldsymbol{A}$, it holds that $a \overset{X}{\sim} \text{intro}(a, \omega)$ for any $X \subseteq \boldsymbol{X} \setminus \{\text{LP}\}$.*

Proof. Let the bisimulation relation be defined as:

$R(X) = \{(((L, \Sigma, \Delta, l), v), (\text{intro}((L, \Sigma, \Delta, l), \omega), v)) \mid (L, \Sigma, \Delta, l) \in \boldsymbol{A}, v(\text{LP}) = \omega(l)\}$

It is trivial to see that each transition can be mimicked and the resulting state is again in $R(X)$. □

5.4.2 Superposition of locations

The next step is the superposition of the locations. The superposition of the locations, which will henceforth be called deflation, is defined as:

$$\text{deflate}((L, \Sigma, \Delta, l)) = (\{k\}, \Sigma, \{(k, g, \sigma, u, k) \mid (l_s, g, \sigma, u, l_t) \in \Delta\}, k)$$

Again, the function is naturally extended to models.

Lemma 8. *Given a set of automata $A \subseteq \boldsymbol{A}$, let $\psi_L = \bigcup_{(L,\Sigma,\Delta,l) \in A} \{(l, k) \mid l \in L\}$ and $\psi_\Sigma = \text{id}$. It holds that $A \leftrightarrows \text{deflate}(A)$. Note that the k in ψ_L is the superimposed state of that specific automaton.*

Proof. A witness is given by:

$$\psi_\Delta = \{((l_s, g, \sigma, u, l_t), (k, g, \sigma, u, k)) \mid (l_s, g, \sigma, u, l_t) \in \boldsymbol{\Delta}\}$$

Again using the assumption that each edge is non-redundant, this is the largest witness with respect to A and A'. Namely, it is clear that, again, no edges are introduced or removed: only the locations are changed and there is a strict equality between the guards and updates. □

Lemma 9. *Given an automaton $a \in \text{intro}(\boldsymbol{A})$, it holds that $a \overset{X}{\sim} \text{deflate}(a)$ for any $X \subseteq \boldsymbol{X}$.*

Proof. Let $R(X) = \{((a, v), (\text{deflate}(a), v)) \mid a \in \text{intro}(\boldsymbol{A})\}$.

First, assume that $(((L, \Sigma, \Delta, l), v), ((\{k\}, \Sigma, \Delta', k), v)) \in R(X)$ and there is a transition $((L, \Sigma, \Delta, l), v) \xrightarrow{\sigma} ((L, \Sigma, \Delta, l'), v')$ for some l', v' and event σ. For this transition, there must be an edge $(l, g, \sigma, u, l') \in \Delta$. Based on the definition of deflation, there must also be an edge $(k, g, \sigma, u, k) \in \Delta'$. Hence, the deflated automaton can make a transition

$$((\{k\}, \Sigma, \Delta', k), v) \xrightarrow{\sigma} ((\{k\}, \Sigma, \Delta', k), v')$$

Secondly, assume there is a transition $((\{k\}, \Sigma, \Delta', k), v) \xrightarrow{\sigma} ((\{k\}, \Sigma, \Delta', k), v')$ for some σ and v'. For this transition, there must be an edge $(k, g, \sigma, u, k) \in \Delta'$. Per definition, there must be an edge $(l, g, \sigma, u, l') \in \Delta$. Hence, the original automaton can make a transition

$$((L, \Sigma, \Delta, l), v) \xrightarrow{\sigma} ((L, \Sigma, \Delta, l'), v')$$

Clearly,

$$(((L, \Sigma, \Delta, l'), v'), ((\{k\}, \Sigma, \Delta', k), v')) \in R(X)$$

□

5.4.3 Superposition of dynamics

The last step of the process is the superposition of the dynamics. To this end, the $\otimes$ operator is introduced:

$$(L, \Sigma, \Delta, l) \otimes (L', \Sigma', \Delta', l') = ((L, \Sigma, \Delta \otimes \Delta', \alpha), (L, \Sigma' \setminus \Sigma, \Delta'', \alpha))$$

The set of transitions for the leftmost automaton of the tuple is defined, as follows:

$$\begin{aligned} \Delta \otimes \Delta' = {} & \{(l, g, \sigma, u, l) \in \Delta \mid \sigma \in \Sigma \setminus \Sigma'\} \cup \\ & \{(l, g \wedge g', \sigma, u \odot u', l) \mid \sigma \in \Sigma \cap \Sigma', (l, g, \sigma, u, l) \in \Delta, (l', g', \sigma, u', l') \in \Delta'\} \end{aligned}$$

The set of transitions for the rightmost automaton of the tuple is defined, as follows:

$$\Delta'' = \{(l, g, \sigma, u, l) \in \Delta' \mid \sigma \in \Sigma' \setminus \Sigma\}$$

Lemma 10. *Given a set of automata $A \subseteq \text{deflate}(\boldsymbol{A})$ and two automata $b, c \in \text{deflate}(\boldsymbol{A})$ such that $b, c \notin A$ and $b \neq c$, let $\psi_L = \text{id}$ and $\psi_\Sigma = \text{id}$. It holds that $A \cup \{b, c\} \leftrightarrows A \cup \{b', c'\}$ with $(b', c') = b \otimes c$.*

Proof. A witness is given by:

$$\begin{array}{rcl} \psi_\Delta & = & \{(\delta, \delta) \mid a \in A, \delta \in a.\Delta\} \\ & \cup & \{(\delta, \delta) \mid \delta \in b.\Delta \cup c.\Delta, \delta.\sigma \notin b.\Sigma \cap c.\Sigma\} \\ & \cup & \{(\delta, \delta \otimes \delta'), (\delta', \delta \otimes \delta') \mid \delta \in b.\Delta, \delta' \in c.\Delta, \delta.\sigma \in b.\Sigma \cap c.\Sigma, \delta.\sigma = \delta'.\sigma\} \end{array}$$

with

$$(l, g, \sigma, u, l) \otimes (l, g', \sigma, u', l) = (l, g \wedge g', \sigma, u \odot u', l)$$

□

Lemma 11. *Given two automata $a, b \in \text{deflate}(\boldsymbol{A})$, it holds that $a \parallel b \overset{X}{\sim} c \parallel d$ with $(c, d) = a \otimes b$.*

Proof. Let $R(X) = \{(a \parallel b, v), (c \parallel d, v) \mid (c, d) = a \otimes b\}$. In this proof the non-synchronous transition are disregarded, as it is trivial to see that each non-synchronous transition can be mimicked.

Assume that $((a \parallel b, v), (c \parallel d, v)) \in R(x)$ and that for a valuation v and (synchronous) event σ there is a transition $(a \parallel b, v) \xrightarrow{\sigma} (a \parallel b, v')$. There must be an edge (l, g, σ, u, l) in automaton a and an edge (l, g', σ, u', l) in automaton b such that

$$v \models g \wedge g' \text{ and } v \cup v' \models u \odot u'$$

From the definition of $\Delta \otimes \Delta'$, it is known that automaton c contains the edge

$$(l, g \wedge g', \sigma, u \odot u', l)$$

for which it also holds that

$$v \models g \wedge g' \text{ and } v \cup v' \models u \odot u'$$

As a result, the transition $(c \parallel d, v) \xrightarrow{\sigma} (c \parallel d, v')$ can occur.

Now assume that the transition $(c \parallel d, v) \xrightarrow{\sigma} (c \parallel d, v')$ occurs for a valuation v and (synchronous) event σ. Only automaton c contains former synchronous transitions, and all are of the form defined by $\Delta \otimes \Delta'$. Therefore, it is known that automaton c contains the edge

$$(l, g \wedge g', \sigma, u \odot u', l)$$

such that

$$v \models g \wedge g' \text{ and } v \cup v' \models u \odot u'$$

Using the definition of $\Delta \times \Delta'$, automaton a must contain an edge (l, g, σ, u, l) and automaton b an edge (l, g', σ, u', l). As it holds that

$$v \models g \wedge g' \text{ and } v \cup v' \models u \odot u'$$

the transition $(a \parallel b, v) \xrightarrow{\sigma} (a \parallel b, v')$ can occur. □

Note that the order of the automata is of importance on the result of the $\otimes$ operation, as discussed earlier; a choice remained for the end-user where to place the resulting dynamics. This will be reflected in the following section, in which an algorithm is provided to remove the synchronous behavior.

5.5 Algorithm

This section provides an algorithm for the removal of synchronous behavior. Provided a set of automata B that model the system when placed in composition, the algorithm is defined as: As stated, some choices remained which result in slightly different

```
**Require:** $A$ = deflate(intro($B$))
 1: $A'' = \varnothing$
 2: while $A \neq \varnothing$ do
 3:     $a$ = choose($A$)
 4:     $A = A \setminus \{a\}$
 5:     $A' = A$
 6:     while $A' \neq \varnothing$ do
 7:         $b$ = choose($A'$)
 8:         $A = A \setminus \{b\}$
 9:         $A' = A' \setminus \{b\}$
10:         $(a, b) = a \otimes b$
11:         $A = A \cup \{b\}$
12:     end while
13:     $A'' = A'' \cup \{a\}$
14: end while
```

outcomes. These choices are reflected by means of the choose operator which denotes that based on the user-provided priorities a specific element from the set is chosen. A visual representation of the inner loop, after an automaton a is chosen, is represented in Figure 5.10.

Lemma 12. *The provided algorithm terminates.*

Proof. In the outer (while) loop, the condition is based on the emptiness of set A. It is clear that during each iteration, an element from this set is removed (line 4). Note that the removal and addition of element to A at lines 8 and 11, respectively, do not influence the number of elements of set A.

For the inner (while) loop, the condition is based on the emptiness of set A', which is initially equal to the set A. It is clear that each iteration an element is removed from the set (line 9).

As a result, both the inner loop and the outer loop terminate and, as a result, the algorithm will terminate. $\square$

Lemma 13. *The provided algorithm is sound.*

Proof. At line 3, a specific element of the set A is taken. For this automaton, each other element within the set A is visited in the inner loop. Using the result of the $\otimes$, automaton a is enriched at each iteration with the combined dynamics of the synchronous transition while automaton b only contains transitions with non-synchronizing events. At the end of the inner loop there is no automaton in A that contains events in its alphabet that are part of the alphabet of a. Hence, all the synchronous behavior of the alphabet of automaton a has been considered and it does not have to be considered again. As this holds for any automaton chosen at line 3, the algorithm is sound. $\square$

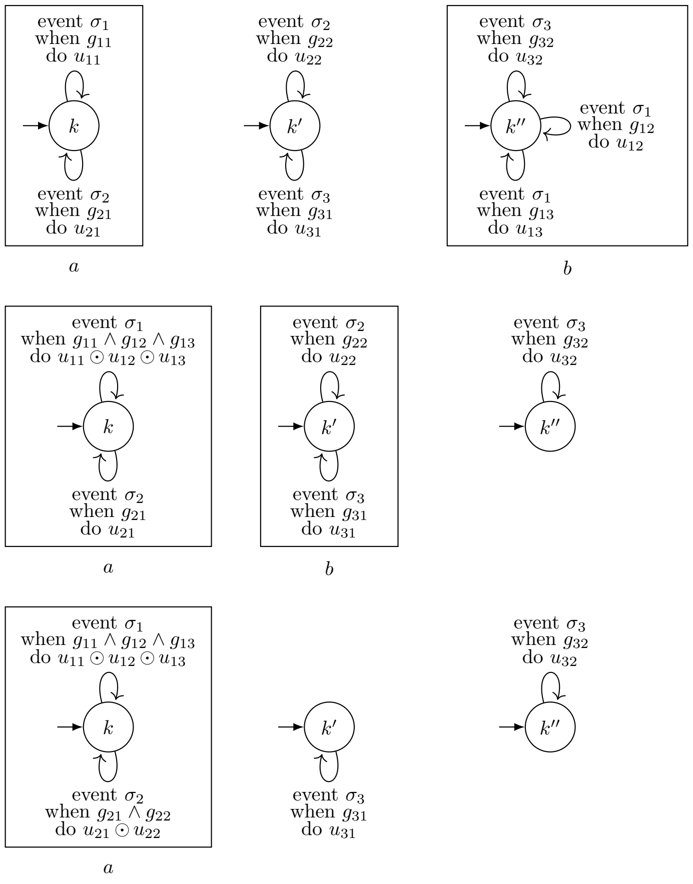
5.6 Concluding remarks

In this chapter, an alternative algorithm for the removal of synchronization has been introduced. Key features of this algorithm are the fact that the resulting model is traceable with respect to the original model, the resulting model is in bisimulation with the original model, and duplication of non-synchronous events is prevented.

Three mathematical operations have been introduced that are needed for the removal of synchronous behavior: the vocalization of locations, the superposition of locations, and the superposition of dynamics. For each of these operations, it has been proven there is a traceability relation between the original and the resulting model, and that the resulting model is in bisimulation with the original model.

An algorithm has been provided that uses these three operations to transform the complete model into a model without synchronization. Using the transitivity results of Chapter 4 it is know that the result of the algorithm is traceable with respect to the original model and is in bisimulation with the original model. The algorithm allows for specific end-user choices where to move the resulting synchronous dynamics.

In the introduction of this chapter, adaptability was briefly mentioned. It would be beneficial to show the difference in adaptability between the standard synchronous product and the result of the algorithm of this chapter, given a (user-specific) definition of adaptability. This will give insights for which cases the approach of this chapter is beneficial and for which cases it is not.



event σ_1 when g_{11} do u_{11}
k
event σ_2 when g_{21} do u_{21}
a
event σ_2 when g_{22} do u_{22}
k'
event σ_3 when g_{31} do u_{31}
event σ_3 when g_{32} do u_{32}
k''
event σ_1 when g_{12} do u_{12}
event σ_1 when g_{13} do u_{13}
b
event σ_1 when $g_{11} \wedge g_{12} \wedge g_{13}$ do $u_{11} \odot u_{12} \odot u_{13}$
k
event σ_2 when g_{21} do u_{21}
a
event σ_2 when g_{22} do u_{22}
k'
event σ_3 when g_{31} do u_{31}
b
event σ_3 when g_{32} do u_{32}
k''
event σ_1 when $g_{11} \wedge g_{12} \wedge g_{13}$ do $u_{11} \odot u_{12} \odot u_{13}$
k
event σ_2 when $g_{21} \wedge g_{22}$ do $u_{21} \odot u_{22}$
a
k'
event σ_3 when g_{31} do u_{31}
event σ_3 when g_{32} do u_{32}
k''

Figure 5.10: Schematic representation of the inner loop of the algorithm, after an automaton a is chosen.

Finally, it may be beneficial to investigate the implications on the model for specific choices on the placement of the synchronous dynamics. It may be the case that the quality and/or flexibility of the model deteriorates for specific choices of placement.

Chapter 6

Real-time implementation

The automatic generation of code from models is an important step in the MBD approach. It reduces the effort of manually coding the controller, while the quality of the code is guaranteed; if a mature generator is used, no coding mistakes are to be expected. Because there are differences in the semantics of a CIF model and arbitrary control platforms, this chapter focusses on how those CIF-specific constructs can be translated to a PLC platform, as this platform is commonly used for BHSs. Note that most of the considerations can be easily extended to other formalisms, e.g., C or Java. Therefore, the code fragments provided in this chapter are given in pseudo-code fitting PLC, C, and Java code. It is clearly stated when and how considerations can be extended to another formalism. If nothing is mentioned, only the PLC is considered. In the end, some specific implementations are provided to cope with the boundary conditions of BHSs. These specific implementations are only valid for a specific subset of CIF models.

6.1 Introduction

In order to generate code that executes in a similar manner as the model, it is important to first determine the difference between the execution (semantics) of a CIF model and code running on a PLC. First, the execution of a PLC is explained. Hereafter, an example model is provided that complies to the form introduced in Chapter 5. Based on this model, the differences between CIF and a PLC are highlighted. In the following sections, solutions are provided to bridge the gap between the two formalisms. Note that using the resulting form, i.e., the end result, of Chapter 5 solves the straightforward difference in semantics, i.e., synchronization.

6.1.1 PLC

A Programmable Logic Controller (PLC) is an industrial computer that is robust for working in harsh environments, e.g., dusty. PLC code consists of commonly found entities, e.g., variables, assignments, routines. Different from the execution of code on a PC platform, a PLC intrinsically executes code in a cyclic behavior.

A standard PLC cycle consists of three different phases, given as follows:

(i) Sensing the current state of the system, i.e., the sensor values, and assigning these values to associated variables.
(ii) Sequential, i.e., line-by-line and top-to-bottom, execution of the PLC program.
(iii) Writing of the outputs, i.e., the actuator values, based on the values of the associated variables.

A schematic representation of the cyclic execution with the above phases is given in Figure 6.1. The size of each phase represents the impact on the cycle; the executional phase (ii) normally takes the most time.

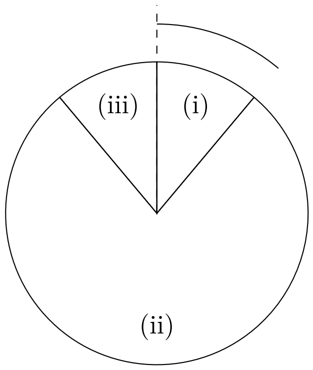


Figure 6.1: Typical PLC cycle.

The time it takes to execute one cycle is called the cycle time. A typical cycle time can be anywhere between 1 and 100 milliseconds. For BHSs, the typical cycle time has to be sought in the order of centiseconds. Commonly, a fixed maximum execution time is specified for the PLC. This entails that the PLC waits with the execution of the next cycle, if the execution of the current cycle was faster than the set cycle time. If the execution time is slower, depending on the settings of the PLC, either the next cycle starts immediately, or the execution stops and an error is triggered. When the PLC has a non-fixed time, it executes a single cycle and immediately starts the next one afterwards. A warning may be issued if the duration of the cycle crosses a certain threshold.

6.1.2 CIF

CIF has a wide variety of modeling concepts. Part of CIF is the ability to allow for the definition of automaton templates, which can be instanced by providing specific arguments to templated parameters. A definition may even be part of another definition. Moreover, CIF allows for the creation of automata groups to gather similar automata together. To allow for a straightforward generation of code, a few of these concepts have been eliminated, such as the concepts of a definition within a definition or a definition within a group.

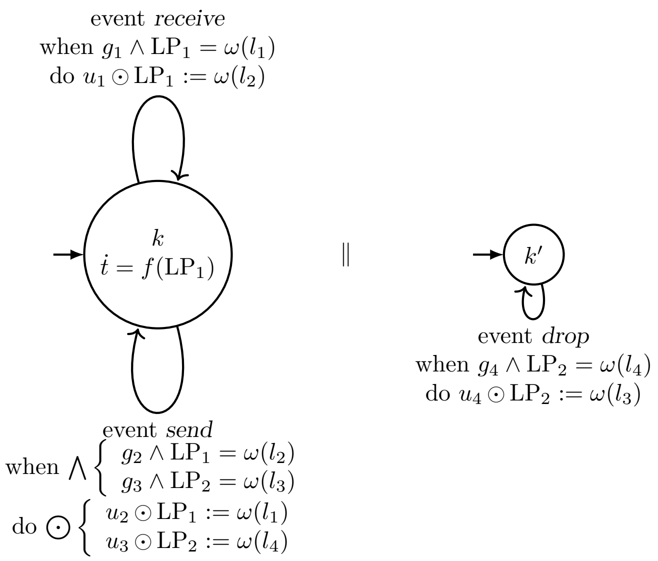

event *receive*
when $g_1 \wedge \mathrm{LP}_1 = \omega(l_1)$
do $u_1 \odot \mathrm{LP}_1 := \omega(l_2)$
k
$\dot{t} = f(\mathrm{LP}_1)$
$\|$
k'
event *drop*
when $g_4 \wedge \mathrm{LP}_2 = \omega(l_4)$
do $u_4 \odot \mathrm{LP}_2 := \omega(l_3)$
event *send*
when $\bigwedge \begin{cases} g_2 \wedge \mathrm{LP}_1 = \omega(l_2) \\ g_3 \wedge \mathrm{LP}_2 = \omega(l_3) \end{cases}$
do $\bigodot \begin{cases} u_2 \odot \mathrm{LP}_1 := \omega(l_1) \\ u_3 \odot \mathrm{LP}_2 := \omega(l_4) \end{cases}$

Figure 6.2: Example of a model ready for code generation.

Considering the semantics of a transition, a transition can occur if the source location is active and the guard is satisfied. As a result of the transition, the valuation is updated according to the update and the target location becomes active. Using the form of Chapter 5 the location is already vocalized, therefore the statement `if g then u end` intuitively describes the semantics of the transition: if the guard `g` is true (incorporating the source location), update `u` is executed (incorporating the change of location). Using this intuition, a preliminary outcome of the generation for the system of Figure 6.2 is given as follows:

```
if g_1 and LP_1 = 1 then
    u_1;
    LP_1 := 2;
end
if ( g_2 and LP_1 = 2 ) and ( g_3 and LP_2 = 1 ) then
    u_2;
    LP_1 := 1;
    u_3;
    LP_2 := 2;
end
if g_4 and LP_2 = 2 then
    u_4;
    LP_2 := 1;
end
```

Note that equations, when present, are disregarded for the moment: the proper translation of equations will be discussed in their associated sections.

A difference in execution between the PLC code and the CIF model can be found in the explicit ordering of execution. Whereas in CIF the model may randomly choose between transitions that are enabled, the order of the PLC code implies a certain order in which the “transitions” are executed. Disregarding the concept of urgency, by generating PLC code a specific subset of the execution behavior is obtained and not all behavior is necessarily present in the code.

Another difference in the execution of the PLC code and the CIF model can be related to the difference in the progression of (model) time (van Beek et al., 2010). In CIF, the execution of a transition does not take any time: the effects of the transition are instantaneous. Moreover, the concept of urgency halts the progression of time as long as transitions are allowed. As a matter of fact, a CIF model may block the progression of time indefinitely. In a PLC, the execution of each statement and each cycle takes a certain (predetermined) amount of time. In other words, time is always progressing.

Finally, a difference may be found in the ordering of updates. In CIF, the updates are defined in terms of the resulting and the current state of the model. The update $x := y$ models the equation $x^+ = y$ in which x^+ is a variable associated with the target state. As a result, $x := y, y := x$, or $x^+ = y, y^+ = x$, models a correct swap of variable values in CIF. In PLC, the assignments are executed in the stated order and the effect on the state is instantaneous. Therefore, it is not possible to have a direct translation of the update $x := y, y := x$; a direct translation does not result in a swap of values but rather a state in which $x^+ = y^+ = y$.

In conclusion, the following differences can be distinguished for the execution of transitions:

	CIF	**PLC**
Execution	Synchronous	Asynchronous
Progression of time	Stagnant	Advances
Ordering	Independent	Dependent

6.2 Progression of time

As a PLC only reads the input and outputs at distinct points in the program, the observed state does not change during the execution of a single cycle. This may be compared to the stagnation of time in a CIF model; as long as no time progresses, the plant model does not change state and, as a result, the observed state also does not change. Note that this is not the case for C and Java. Therefore, the observations made in this section are not directly applicable for C and Java.

To close the gap between the difference in time progression, the execution pattern (or scheme) as shown in Figure 6.3 is proposed. To make a distinction between a PLC execution with and without a fixed cycle time, the process of waiting for the next cycle is added in gray. The variable `done` is used to flag whether or not a transition was executed in the previous cycle. While this is the case, model time cannot progress and all guards must be evaluated again.

Using the scheme of Figure 6.3, the execution of the PLC code will be similar to the CIF model execution as long as the cycle time is smaller than the response

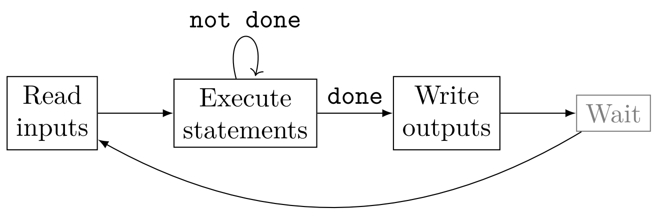


Figure 6.3: Schematic layout of a PLC execution when urgency is implemented.

time of the system; when the system under control behaves faster than the controller, i.e., sensor changes and/or processes always happen faster than the response time of the controller, the supervisor cannot be fitted on the PLC. Therefore, either a faster processor is needed or the code must be optimized when the PLC program is too slow. Optimization of the code is beyond the scope of this thesis.

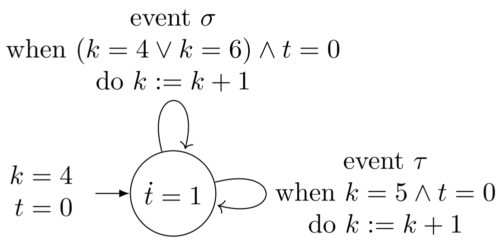


Figure 6.4: An automaton in which urgency plays a role.

To implement the scheme of Figure 6.3, a simple loop (`repeat`) is needed and the introduction of the variable `done`. The value of the variable `done` is set to true at the beginning of the loop and set to false at the end of each transition. Hence, when no transition did occur during the loop the value is true, and the loop may be broken. Schematically, this will result in the following layout:

```
repeat
  done := true
  // event σ
  if /* guard */ then
    // updates
    done := false
  end
  // event τ
  if /* guard */ then
    // updates
    done := false
  end
until done end
```

To illustrate the scheme of Figure 6.3, consider the system of Figure 6.4. In this particular system, urgency is of importance due to the strict equality posed on the time-dependent variable t.

Two different listings are provided below, where the leftmost listing is a naive approach and the rightmost uses the scheme of Figure 6.3:

```
t := t + delta



// event σ
if (k = 4 or k = 6) and t = 0 then
  k := k + 1

end
// event τ
if k = 5 and t = 0 then
  k := k + 1

end
```

```
t := t + delta

repeat
  done := true
  // event σ
  if (k = 4 or k = 6) and t = 0 then
    k := k + 1
    done := false
  end
  // event τ
  if k = 5 and t = 0 then
    k := k + 1
    done := false
  end
until done end
```

Herein, `delta` is the (measured) cycle time of the previous cycle, such that `t := t + delta` is the result of the differential equation $\dot{t} = 1$. Note that `delta` is zero during the first cycle and therefore the model time is indeed calculated after each cycle.

Note the difference in the executional behavior between the two PLC programs with respect to the CIF model. For the leftmost program, the variable `t` is incremented after the first cycle and `t = 0` no longer holds. As a result, the state in which $k = 6$ is never reached in the PLC program, while this state is clearly reached in the CIF model. The rightmost program correctly implements urgency, where the loop is repeated until no transition is enabled anymore. Therefore, after executing the transition labeled τ, the transition σ is executed in the same cycle with the "old" value of t, like in the CIF model.

It is clear that eliminating the concept of urgency induces an additional loop in the system. As a result of this internal loop, the cycle time may fluctuate heavily when a non-fixed cycle time is chosen. Or, the maximum fixed cycle time may be violated. This may be undesired, as the reaction time of the system may become unpredictable. In Section 6.5, the urgency loop is not implemented due to these fluctuations. Note that this has several implications and cannot be done for every system; the extra loop is needed when urgency is to be correctly implemented on the PLC. Clearly, adding the extra loop is one way (as there may be others) to allow systems like Figure 6.4 to be implemented on the PLC with a similar execution as the CIF model.

6.3 Ordering of statements

As stated in the problem definition of this chapter, the update $x := y, y := x$ yields a different resulting in variable values when executed on, for example, a PLC. Whereas CIF swaps the values of the variables, the variables in the PLC both end up with the value of y. Another problem arises with an update of the form $x := 5, y := x$, which results in both x and y having the value of 5 on a PLC. In CIF, the value of y in the target state gets the source state's value of x, and the value of x in the target states becomes 5.

There are two straightforward ways of solving problems arising with the updates semantics of CIF, applicable for all coding formalisms thinkable:

(1) Reorder the assignments.

(2) Introduce one or more temporary variables.

Note that the first solution is not appropriate for all situations, as will be shown. However, it will create less overhead as copying the values to temporary variables is prevented. Note that a mixed solution may also be a solution. Based on the information provided in the following paragraphs, the end-user may choose which of the two solutions may be most beneficial for which variable.

The goal of reordering the variables, if possible, is to prevent any "read-after-write" situations. A dependency graph may be created and Kahn's algorithm may be applied to find an optimal ordering (Kahn, 1962). However, it is immediately clear that for $x := y, y := x$ no acyclic graph exists and therefore no solution. Hence, reordering is not possible in this particular case and one or more temporary variables need to be introduced.

A generic way of introducing temporary variables is to generate a new twin variable and to use this twin variable when doing the assignment. As an example, consider the following solution where two variables x and y are considered:

```
x_ := x
y_ := y
x := x_ + y_
y := 2 * y_ + 3 * x_
```

Note the actual assignments are chosen to have a cyclic dependency.

The easiest way of introducing temporary variables is the introduction of a state structure. This structure contains the values of all state variables. As the complete state is encoded in a structure, it is very simple to make a copy to store the current state. Using two state structures, for instance, state and state_, that, respectively, denote the state before and after the transition, a solution is given as follows:

```
if /* guard */ then
    state_ := state
    state_[ x ] := state[ y ]
    state_[ y ] := state[ x ]
    state := state_
end
```

Here, state[x] is used to denote the value of the variable x contained in the structure state. Note that this approach has a strong resemblance to the way CIF evaluates updates: state models the valuation v while state_ models the valuation v^+.

A downside of using a state structure is the need of copying a potentially large structure. For large systems, the state structure may occupy several megabytes and it may take a couple of milliseconds to copy. As this operation must be done for each transition, it may become very inefficient. A possible way to reduce the amount of data copied is to determine which variables are dependent by using a dependency graph and Tarjan's algorithm (Nuutila and Soisalon-Soininen, 1994). This algorithm finds strongly connected components. Based on the result of this algorithm, dummy variables may be introduced to break the cyclic dependency.

6.4 Implementation of equations

CIF models have two different types of equations. In this section, first the implementation of algebraic equations is considered. Last, the implementation of differential equations is considered.

6.4.1 Algebraic equations

Algebraic equations depend on the current state. Therefore, the value of the associated algebraic variable may change when the state changes. Three very straightforward implementations can be thought of for algebraic variables:

(1) Update the values of the variables according to the algebraic equations after each state change.

(2) Substitute the algebraic variables with their associated equation.

(3) Implement the algebraic equations as functions and evaluate when needed.

The benefits and downsides of each of the three approaches is shown based on the system depicted in Figure 6.5. The placement of the actual algebraic equations is implied by the different methods. The system has two algebraic variables, i.e., y and z, and associated equations. One of the algebraic variables depends on the value of the other algebraic variable. The different implementations are given in Figure 6.6, Figure 6.7, and Figure 6.8. Note that for the implementation of Figure 6.6 it is assumed that initially `z = y * 2` holds.

Whereas the implementation of the algebraic variables, as shown in Figure 6.6, still resembles the original guards, it is clear that the adaptability is low. If one of the algebraic equations would have to change, a lot of changes must be made throughout the code. The same holds for the implementation of Figure 6.7, although here the changes are concentrated to the places where the algebraic variables are used.

The implementation of Figure 6.8 resembles the original edges and it is possible to change the algebraic equations to some degree. However, when new variables are introduced to the algebraic equations, extensive changes must be made to incorporate these changes. For instance, when the equation of z is changed to $y + x$, the signature of the function, i.e., the parameters of the function, must be adapted and therefore all function calls.

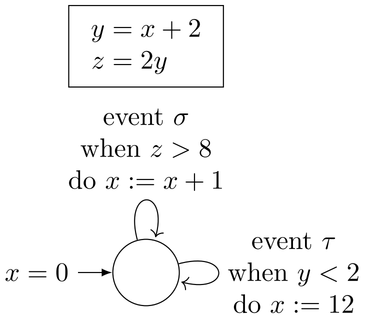


Figure 6.5: A simple automaton with two algebraic variables.

```
// event σ
if z > 8 then
    x := x + 1
    y := x + 2
    z := y * 2
end
// event τ
if y < 2 then
    x := 12
    y := x + 2
    z := y * 2
end
```

Figure 6.6: Implementation of algebraic equations by means of updating after assignment.

```
// event σ
if ( 2 * ( x + 2 ) ) > 8 then
    x := x + 1
end
// event τ
if ( x + 2 ) < 2 then
    x := 12
end
```

Figure 6.7: Implementation of algebraic equations by means of substitution.

6.4.2 Differential

The implementation of differential equations can be done in several ways. The differential equation may be solved beforehand and the solution may be implemented as a function or a simple assignment. In this Ph.D. project, only simple differential equations were encountered and such an approach was possible. As done earlier, consider the very simple differential equation $\dot{x} = 1$. The solution to this equation is trivially $x(t) = t$, assuming the initial condition to be zero. Hence, the solution

```
func y( x ) = x + 2
func z( y ) = y * 2
// event σ
if z( y( x ) ) > 8 then
    x := x + 1
end
// event τ
if y( x ) < 2 then
    x := 12
end
```

Figure 6.8: Implementation of algebraic equations by means of functions.

may be implemented as `x := t` with `t` the (measured) time from the starting of the program.

For the sake of completeness, a solution is also provided for the case a solution does not exist and the differential equation must be solved during runtime. Several solutions exists for solving differential equations, of which the the Runge-Kutta methods (Heath, 2002) are probably best known. The associated solution scheme may be encoded to solve the differential equations during runtime. A straightforward choice for the step size would be the cycle time. However, as the solution of a Runge-Kutta method depends on the chosen step size, a fluctuating cycle time is not beneficial and a fixed cycle time is desired. Nevertheless, a fixed cycle time is not always possible or desired and therefore a solution is needed to solve the differential equation with a fixed step size, while the controller has a fluctuating cycle time.

Luckily, the problem of a fluctuating cycle time and the need for a fixed step size is well-known in the gaming industry. A solution is shown by Nystrom (2014). The idea is to have a specified static time difference between updating the solution of the differential equations. Let **`step`** be the desired step size, let **`delta`** be the current cycle time, and let **`f`** be a function calculating the solution to the differential equation. A possible implementation with a fixed step size is given as:

```
while delta >= step do
    x := f( x, step )
    delta := delta - step
end
```

Note that additional measures have to be taken, beyond the scope of this thesis, when **`delta`** is not exactly divisible by **`step`**, i.e., **`delta`** becomes not exactly zero after the loop is finished.

To be able to solve differential equations with a constant time step, an additional loop is added to the code. A schematic representation is given in Figure 6.9. In gray the loop is added that simulates the urgency concept, and the possible waiting when a fixed cycle time is chosen. Because this extra loop is added on top of a program with minimally fluctuating cycle time, this loop will minimally influence the fluctuations of the cycle time.

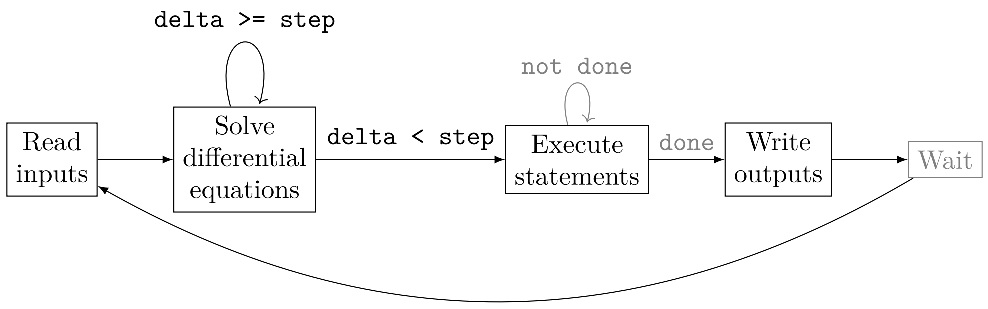


Figure 6.9: Schematic layout of a PLC execution when differential equations are solved on the PLC.

6.5 Code generator

During this Ph.D. project a PLC code generator has been made specifically to cope with the size and boundary conditions of BHSs. In this section, the problems encountered and the solutions for these problems are provided. The biggest change with respect to the general solutions provided in the previous sections is the removal of the urgency loop. Note that none of the proposed solutions in this section are generic; all these solutions have been chosen based on the desired outcome and the allowed flexibility of the models describing the BHSs.

Boundary conditions are posed on the maximum duration of a PLC cycle for a given size of the system. In the models created during this project, a PLC cycle cannot exceed more than a couple of centiseconds. This imposes a constraint on how the ordering of assignments and the algebraic equations are implemented. Namely, the largest system contains approximately 5000 variables. This results in a state structure of roughly 20 kilobytes[6]. Assuming that approximately three quarters of the, in total, 800 transitions assign one or more variables each cycle, 12 megabytes must be copied. Copying this amount of data is impossible within the desired maximum cycle time: the choice was made to determine the dependencies between the assignments and reorder the assignments to break these dependencies, introducing a temporary variable when strictly necessary. This is, as discussed earlier, a hybrid approach in which reordering and introduction of temporary variables are both used. In this particular case, this hybrid approach allowed for the correct implementation of the updates while not increasing the cycle time by avoiding the need to copy the complete state.

As stated, no loop was implemented to correctly implement urgency. The induced fluctuation of the cycle time was one of the reasons, as a difference in execution time can be essential. Namely, products may have traveled an additional 5 millimeters on belts moving with 1 meter per second when fluctuations of 5 milliseconds are encountered. 5 millimeters itself does not sound much, but if each action accumulates these 5 millimeters the final result may be completely wrong.

[6]A mean size of 4 bytes per variable is assumed.

The absence of this loop does not pose any problems as the workings of the controller do not depend on the concept of urgency; there are no automata where the consecutive execution of two transitions is dependent on the concept of urgency, as shown in Figure 6.4, in the BHSs considered for this Ph.D. project. Note that the ordering of the transitions becomes of importance, in some cases, when removing this loop. Details to maximize responsiveness are provided in Appendix B.

To allow a similar structure as the model, it was chosen that each automaton is coded in a so-called function block. Such a function block has input variables, output variables, and internal variables. In the function block, each transition is encoded as an if-statement as explained earlier. As an example, the resulting code for Figure 6.5 may look as follows:

```
func y( x ) = x + 2
func z( y ) = y * 2

block A( in out x )
    // event σ
    if z( y( x ) ) > 8 then
        x := x + 1
    end
    // event τ
    if y( x ) < 2 then
        x := 12
    end
end
```

This pseudo-code defines a function block named `A` with a parameter `x` that is both input and output, comparable to a pointer in C or a reference in Java. As a result, when the state variable `x` is passed to this function, encoded in a different part of the program and not shown in the above listing, the value is updated without assigning it after the call. If other automata are present in the model, additional blocks are added to the code. The naming can be arbitrary, but it is desired to use a name that represents the specification it encodes.

The automaton is executed by calling the block, i.e., `A( x_ );` by calling the function block, the statements contained in the block are executed. The value of `x` depends on the current value of `x_`. Afterwards, the resulting value of `x` is returned and written to the argument `x_`.

6.6 Applications

To show the application of the modeling and validation techniques, as provided in this thesis, three large test cases have been conducted at Vanderlande Industries, Veghel, The Netherlands. In the first project, a controller has been created for a large BHS consisting of 45 sections (4 distinct types), 9 zones (4 distinct types), and 1 area, with an approximated length of 120 meters. Approximately 150 automata are used to model the software components and 60 automata are used to model the hardware components. In this project it was of the essence to show that indeed the quality of the controller could be determined before the actual controller was built (or generated in this case). The complete project was finished in 9 months. Moreover, where normally a complete team works on the creation of the controller, only 2-3 people were working fulltime on this project.

During the project, a large number of requirements from the product books of the actual system have been implemented. These requirements include error detection and handling for situations such as product collisions and the illegal placement and removal of products. Furthermore, they include functional requirements such as energy save, routing, and support of different operation modes. Following the product book, models were made for each desired section, zone, or area. This was done hierarchically, as discussed in Chapter 1, starting from the sections. For each component, e.g., a TRS, the hardware and software model were made simultaneously. In such a fashion, the interface between them could be optimally designed. Moreover, validation of the components could be done immediately (early integration) and errors could be solved before, for instance, zone functionality was added.

In more detail, each component was first modeled with a select (or reduced) set of features. For instance, the first version of a TRS was very simple; the motor was always running and only one product could be present any momemt in time. Using the interactive simulations, supported by a visual representation of the automata models themselves, the behavior was validated and cross-checked with Vanderlande Industries; meetings were held to allow feedback from engineers. When a model was considered correct, additional features were added until a model with the desired set of features emerged. In such a way, errors could be detected early on in the design process and, therefore, could easily be solved. Gradually, the models converged to their fully featured versions.

After extensive validation of the complete model, code was generated using the approach of Swartjes et al. (2014). This allowed us to test the controller on the actual hardware. This is a crucial step, as it was key to show that indeed the quality increased. Hence, fewer (or no) errors should emerge during these tests. The resulting supervisory controller of the BHS runs on a PLC.

During the testing phase some errors did emerge, which could be categorized in four types:

Generation errors	Errors found in the code generator.
Hardware errors	Errors due to hardware failures or discrepancies.
Abstraction errors	Errors due to simplifications of the hardware model.
Unwanted behavior	Behavior that is undesired from an implementation point of view.

Generation errors In this project, the code generator was used for the first time. Therefore, errors were to be expected. However, all of these errors could be directly associated with the generation tool and not with the model itself.

The smaller errors that were found, and fixed, were mainly due to naming conventions and differences in case sensitivity. In the CIF model, reserved keywords of PLC code could be used, which results in errors when code is generated. Larger errors were found due to the difference between the semantics of CIF and the semantics of a PLC. In these cases, the controller on the PLC behaved completely differently than any of the simulations. Many of these errors could be traced back to the ordering of assignments and urgency issues (van Beek et al., 2010). All errors that were encountered were fixed by adapting the generator, if possible. Some problems could not be solved at the moment in the generator, e.g., the optimal ordering of assignments; during this specific project, no heuristics was present in the generator to automatically solve transition ordering. Therefore, these problems were directly solved by adapting the model.

Hardware errors In this project two hardware errors were encountered. Both errors were due to discrepancies between the system specification and the actual hardware. The first error was due to a failing hardware component. This can only be solved by replacing the part and is not associated to the model in any way.

The second error was due the absence of a hardware component: a PEC was unknowingly removed from a TRS to be used as a spare part for another system, although it was specified that all TRSs would contain (at least) one PEC. Since our model depends on the presence of a PEC, errors where generated and the system could not be started. As we did not consider the requirement on the presence of PECs, this was not a modeling error. As the part could not be replaced, it was chosen to adapt the model in such a way that the PEC was not necessary for this TRS. Due to the modular structure of the system, this could be done quite easily.

Abstraction errors In general, a model is an abstraction of reality. For instance, friction is not modeled and the change in velocity is assumed to be instantaneous. Therefore, errors could be introduced due to, for instance, slip and/or finite acceleration/deceleration. These errors are expected when testing the controller on a real system. These are indeed modeling errors, but they are not related to the quality of the model; the requirements are correctly implemented under the given assumptions.

Nevertheless, there are several ways to prevent these errors by adapting the model. Firstly, it is possible to incorporate these abstracted phenomena. However, this could potentially lead to a decrease in understandability of the model as a result of increased complexity. Secondly, parameters can be introduced related to friction, i.e., timing and velocity constants that can be adapted afterwards.

In our model, the latter approach was taken to prevent this increase in complexity. These parameters are also translated to the resulting controller code. By tweaking these parameters in the code during tests, the errors associated to these abstraction errors could all be resolved. For maintainability of the code it is essential that the final parameter values are documented somewhere for future reference, as these tweaks will break the link between model and code. However, as the model does not contain the same real-world dynamics, the parameters cannot be pushed back to the model.

Unwanted behavior In this project, one of the requirements with respect to the interaction of products on a single TRS was implemented precisely as stated in the product book. Because of this exact implementation, it could occur that the motor would start and would immediately stop. This behavior was observed due to audible feedback; the starting and stopping of the motor makes a clicking sound. Although this behavior is exactly what the requirements state and therefore not an error per se, it is undesired from an implementation point of view: frequently starting and stopping the motors induces a high(er) wear and tear. Hence, adaptations were made in the motor controller: a form of slack was introduced to the controller that prevented rapidly switching the motor signal.

After the errors were addressed, no behavior was observed that differed from the simulations. Based on this criterion, the controller was deemed of sufficient quality. It was never necessary to adapt the models as a result of modeling errors. Hence, it can be concluded that due to the combination of proper modeling and extensive validation, (potential) modeling errors were already solved before testing: the quality of the controller could indeed be assessed before the controller was built. As a result, a shorter time was needed to test the controller on the actual hardware, which in its turn reduces the time to design a controller. This reduction in the effort of creating supervisory controllers for BHSs is one from which the industry can benefit greatly.

The second and third project both comprised the recreation of a real airport with far greater functionality than the first project. During these projects, numerous additional requirements from the product book, and additional sections and zones were created. The third project also focussed on inter-area communication, or inter-PLC communication. The model of the considered system consists of approximately 600 automata, 800 events, and 5000 variables. As a result, this system is almost four times the size of the system of the first project. Due to confidentiality, detailed information on the layout and workings cannot be provided. However, all modeling, validation, and generation techniques of this thesis could also be easily applied to this system. Due to the reuse of a lot of components, a first working prototype, based solely on the layout and not on the functionality, could already be made in a manner of weeks. As the system could be scaled up quite easily, the flexibility of the model was also proven.

Because of the impossibility to test the controller on the actual hardware of the airport, the focus was on emulation-based testing after the controller was generated. Emulation is a form of testing were the controller runs on a PLC, but the hardware is simulated on a computer. The emulation model, i.e., a model of the hardware, was kindly provided by Vanderlande Industries. Ideally, the hardware model is reused for this purpose. However, no interface was present to create this linkage.

Finally, the code generator was extended to incorporate a new backend. This was done to show the added flexibility of generating code for different platforms from the same model. Since only the PLC version could be tested, the correctness of the code was assessed by visually inspecting the code and testing for compilation errors. In the earlier stages, compilations errors did emerge. Based on the compilation errors, the generator was improved. Finally, the generator was able to generate code which compiled without any errors.

Code generation errors Since new concepts were used in the latter two projects, the code generation had to be extended. This led to some new generation errors, that were fixed during the course of the second project. During the third project, due to the inter-PLC communication, additional errors as a result of an incorrect ordering of assignments were found in the code generator, and were also fixed.

Hardware errors Some errors emerged due to incompatibilities between the hardware model used in CIF and the hardware model used during emulation. In the emulation model, a section needs a constant signal from its parent zone to determine whether or not the system is "healthy". These errors were quickly fixed by extending the hardware model in CIF to resemble the emulation model.

Abstraction errors and unwanted behavior No errors were found during these projects that can be classified as abstraction errors or unwanted behavior.

The results and insight obtained during these projects have shown Vanderlande Industries the potential of adopting a formal model-based design approach. These projects were an indicator to Vanderlande Industries how they can adopt and further improve MBD for their software components with formal modeling methods. As a result, projects have been started that will adopt a formal MBD approach using proven modeling formalism methods, and covering the entire software development process from requirements up to and including code generation.

6.7 Concluding remarks

As is shown in this chapter, a CIF model can be transformed to code capable of running on a PLC. In order to create the code, concepts of CIF not available on the control platform need to be either eliminated or simulated. A big difference between CIF models and a PLC is the progression of time and the ordering of statements. To cope with the difference in the progression of time, an additional loop needs to be added. To cope with the ordering of statements, temporary variables may be introduced or, when possible, the ordering of the statements may be changed.

In addition to the provided generic solutions, additional measures needed to be taken to allow the code generation for BHSs. During three different test cases, code was successfully generated and controllers were obtained that appeared to behave as modeled. During these three projects some errors emerged due to the code generator, some due to the hardware, some because of the abstractions used in the model, and some undesired behavior. These errors were solved during the course of the project. No errors were found due to wrong or wrongly implemented specifications. This shows the benefit of using early integration. When a more matured generator is used it is likely that no generation errors are obtained anymore. Hence, code generation indeed reduces the effort and increases quality. Finally, it is very straightforward to see that supporting different control platforms becomes a simpler task with code generation.

Chapter 7

Conclusions

In the introduction, four challenges were identified for the creation of supervisory controllers for BHSs:

(1) How to exploit MBD and early integration for supervisory controllers (for BHSs) to achieve models with an increased quality and flexibility and a reduced complexity?

(2) How to provide user-feedback on the removal of states during synthesis in order to increase the usability of supervisory control synthesis in industry?

(3) How to perform model-to-model transformations, while retaining full traceability information on the requirements?

(4) How to generate code from models such that the execution of these models on a real-time platform is similar to the validated behavior?

In the introduction it is explained that BHSs are unique systems due to their size, structure, and partial observability. Creating supervisory controllers for BHSs is therefore challenging. To aid the design of a supervisory controller, guidelines are introduced to support the creation of supervisors with an increased quality and flexibility and reduced complexity.

In addition to these guidelines, it is shown that allowing the creation of both models for the uncontrolled system and the controller, errors could be pinpointed early on in the design process. Because errors can be found earlier, the errors will have a smaller impact, which results in shorter design cycles.

An alternative approach to modeling the supervisory controller is the usage of supervisory control synthesis. This may further reduce the number of design cycles needed as it allows for controllers that are correct by design. Although synthesis has the potential of decreasing design time, uptake in industry is low due to, among others, lack of user-feedback. In this thesis a formal framework is introduced that allows for the generation of causes, which provide an explanation on the absence of a specific state in the supervised system.

Using the derived formal system of rules, an algorithm for the derivation of causes was created that is part of the synthesis algorithm. During synthesis, information on the absence of states is stored such that it can be employed afterwards to construct causes. Although not all causes can be created/found using the algorithm, it allows for a direct feedback to the user at the end of synthesis.

After the model of the system has been created, other tools may be used to verify the controlled behavior or code may be generated from the controller model. To this end, it may be necessary to perform model-to-model transformations. However, information on the requirements may be lost due to these transformations. To allow for a linkage between the original model and the resulting (transformed) model, a relation on the syntactical level of these models is defined. With this relation, transitions related to specific requirements can be traced from the original model to the transformed model and vice versa.

Although synchronization is useful when modeling systems, namely synchronization is used to increase the quality and flexibility of models, it is chosen in this thesis to remove the synchronous behavior when code is generated. Removing the synchronous behavior from the model in the traditional way results in the duplication of non-shared events, which lead to code duplication. Therefore, an alternative for the removal of synchronous behavior has been introduced that does not suffer from this duplication. Moreover, it is proven that the resulting model is traceable with respect to the original model and is in bisimulation with the original model.

Using the alternative algorithm for the removal of synchronization only removes synchronization. However, CIF has more concepts that may lead to differences in executional behavior when not properly transformed. These differences have been identified and solutions have been provided for the generation of code for CIF models.

In addition, a specific code generator has been created for models of BHSs to cope with the strict (timing) requirements of the controller code. These changes were possible due to the structure of the BHS controllers; in general, these changes will result in behavior different from the original CIF model. Using this generator, PLC code was generated for three large test cases, of which one a real airport. Using this code generator it was shown that the resulting behavior complies to the behavioral requirements, with the aid of visual inspection.

This thesis focuses on model-based design of supervisory controllers for BHSs. However, a BHS is not the only system that fits the methods introduced in this thesis. The concepts and algorithms provided in this thesis can, in principle, be applied to the model-based design of a wider range of systems, e.g., manufacturing lines and MRI-scanners, though some alterations may be necessary.

When a model-based design approach is adopted, the effort of creating supervisory controllers tends to decrease. Intuitively, the effort is decreased due to early integration, by using code generation, and synthesis (together with the generation of causes). Namely, errors are found when they have less impact, coding errors are eliminated and the controller is safe, nonblocking, and controllable by design.

With respect to the definitions provided in the introduction, it can be stated that the model-based design of supervisory controllers for BHSs will result in an increase of quality and flexibility, while complexity is reduced. This was highlighted during the testing phases of the three projects conducted at Vanderlande Industries. Errors found when the generated code ran on real-time platforms were mostly due to interpretation errors or code generation errors; behavioral errors would in most cases already have been detected and solved using simulation or other validation techniques.

Due to the industrially critical nature of this work, not all information could be disclosed on the different cases we have worked on. This entails that it is very hard to capture the true complexity of the considered systems in this thesis. This becomes evident in both Chapter 2 and Chapter 6. In Chapter 2 the full complexity of the models cannot be disclosed as this would directly give insights in the control requirements: parts of the behavior of the real-life system, classified competitor-sensitive, have been removed in this thesis and a more simplistic view of the behavior and/or requirements is presented. As an example, the detection of blocked products comprises numerous requirements and not solely the requirement provided in this thesis. In Chapter 6 no information could be disclosed on the real airport used for code generation. For this airport a significant set of the requirements was implemented, comprising the transportation and routing of products and even communication with SCADA-like systems.

For the last year I have been working as an MBD engineer at Vanderlande. At Vanderlande, a suite of tools is used providing similar means of validation and code generation as discussed in this thesis. However, formal verification and synthesis are not yet part of this package.

Currently, MBD is used for feasibility studies, i.e., to assess if the proposed system can work, as well as code generation, i.e., product development. There is a wide variety of projects going on, including MBD for airport solutions. I have been working on various projects for parcel and postal purposes, e.g., automated inducting of parcels on a sorter. As an MBD engineer it is my job to model complete system components, i.e., plant and controller, that respect the provided requirements. Or, when no requirements are (yet) specified, to model systems that assess the feasibility.

Validation of these models is mostly done in cooperation with other disciplines; using visualization techniques to show the (modeled) behavior proves to be very helpful. Until now I have been able to successfully apply some of the results obtained during this Ph.D. project. However, fully-fledged BHSs are extremely challenging for supervisory controller synthesis and verification, because essential control requirements for such systems are inherently of a combined continuous-time/discrete-event nature, dealing with, among others, product distances on moving conveyer belts. This leads to state-space explosion even for small systems. Nevertheless, the modeling consideration put forward in this thesis in Chapter 2 regarding flexibility have been extensively applied: the models I have created use the concept of an observer, use features to enable or disable behavioral requirements, and are equipment-based.

Although MBD is still relatively new for Vanderlande, the success of these projects shows how MBD can help to speed up future projects, increase quality, and mitigate risks. As a result, an increasing interest is shown in MBD at Vanderlande.

Appendix A

Remaining challenges for supervisory control reasoning

In this chapter, some challenges and difficulties are highlighted that must be overcome before an algorithm can be provided that returns all "relevant" causes for every state removed during synthesis; note that there may be infinitely many causes for the removal of a specific state and it is not our goal to provide every cause. It is shown that relevant is a difficult concept. Therefore, developing an algorithm that provides all relevant causes is an open research question.

A.1 Cycles

Due to the potential presence of cycles in a system, the number of paths leading from a state to the set of marked states or leading from the set of initial states to a state may be infinite. As the rules related to blocked and unreachable states depend on these sets, it is important to investigate whether this set can be reduced to a finite set. For an example of a system with a cycle, consider the system depicted in Figure A.1.

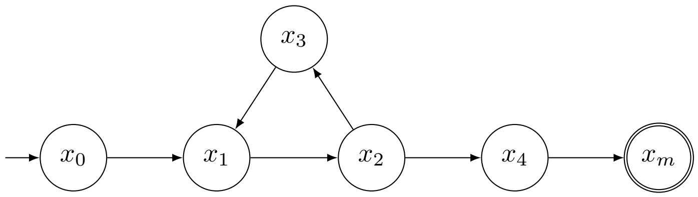


Figure A.1: Infinite number of paths from an initial to an marked state, due to the presence of a cycle.

State x_4 has an infinite number of paths from the initial state to itself, while state x_0 has an infinite number of paths from itself leading to the marked state. In an naive algorithmic implementation of the deduction rules for the derivation of the complete set of causes, whatever that may be, it is impossible to provide all reasons fore blockingness or unreachability of a state as it is therefore necessary to visit all states along all paths of which there are an infinite number. To overcome this problem, the concept of a reduced path is defined.

Definition 12 (Reduced path). *Given a path with zero or more cycles, e.g.,*

$$x \longrightarrow x' \,[\longrightarrow \ldots \longrightarrow x']^{\star} \longrightarrow \ldots \longrightarrow x''$$

a reduced path is the subpath containing no cycles, i.e.,

$$x \longrightarrow x' \longrightarrow \ldots \longrightarrow x''$$

The notation $P'(x, x')$ is used to denote all reduced paths starting in state x and reaching state x'. Note that such a set can always be determined.

In Figure A.2, the reduced path from x_0 to x_m is denoted by the thick states and transitions.

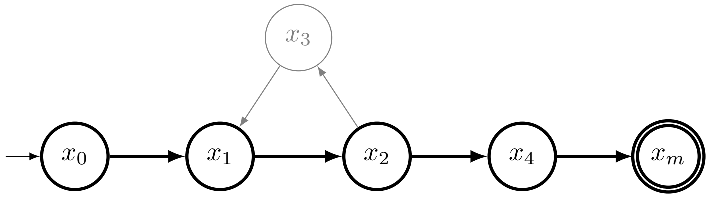


Figure A.2: Schematic overview of the reduced path.

Lemma 14. *Given a transition system $(X, \Sigma, \longrightarrow, X_0, X_m)$, the blockingness of a state $x \in X$ can be proven by only considering reduced paths.*

$$\frac{\text{gone}(x_1) \quad \cdots \quad \text{gone}(x_n)}{\text{gone}(x)} \; \textit{for each set } \{x_1, \cdots, x_n\} \subseteq X \textit{ such that}$$
$$\{x_1, \cdots, x_n\} \cap p \neq \varnothing \textit{ for any } p \in P(x, X_m)$$
$$\Longleftrightarrow$$
$$\frac{\text{gone}(x'_1) \quad \cdots \quad \text{gone}(x'_n)}{\text{gone}(x)} \; \textit{for each set } \{x'_1, \cdots, x'_n\} \subseteq X' \textit{ such that}$$
$$\{x'_1, \cdots, x'_n\} \cap p' \neq \varnothing \textit{ for any } p' \in P'(x, X_m)$$

Proof. First, assume the left-hand side. As each path in P' is also a path in P it is trivial that the right-hand side holds.

Finally, assume the right-hand side. Consider an arbitrary path from P. If that path has at least one cycle, a subset of states from this path is contained in the reduced version of that path that is contained in P'. As we can prove gone(x) using the reduced path, we can trivially prove gone(x) for any path that can be reduced to this reduced path. $\square$

A.2 Minimality

The derivation of causes is sound and complete, as proven. However, this does not imply that all derived causes are usable: the size of a cause may be so large that it is no longer comprehensible, defeating its purpose. As a result, the question arises whether it is possible to determine a minimal cause.

A possible way of looking at minimality is with respect to the depth of the tree. A cause can be considered smaller than another cause if it is of lesser depth but yields the same conclusion for the same premisses. Visually, this is presented in Figure A.3 where the rightmost cause is smaller than the leftmost cause. Namely, both causes derive the same reasons while the depth of the rightmost tree is less than the leftmost tree. Do note that different information may be present in the two causes, as the leftmost tree consists information about w that is non-existing in the rightmost cause.

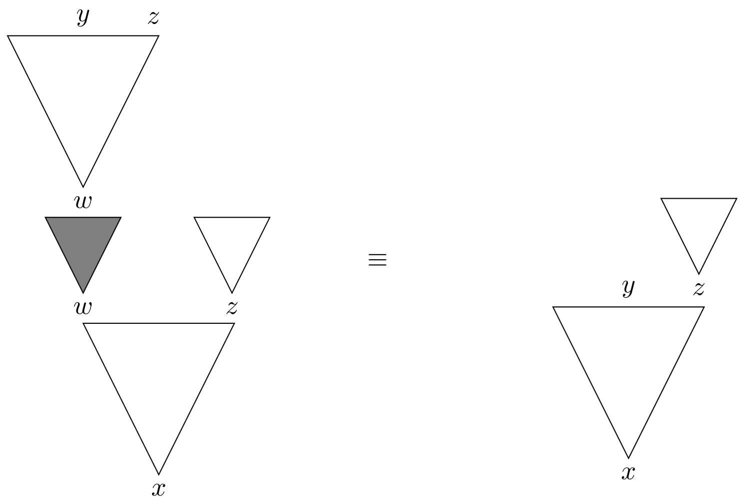


Figure A.3: Two causes: the rightmost cause is smaller than the leftmost cause.

The minimal cause is then the smallest of them all. Although visually it is easy to show the meaning of minimality, the semantical part of minimality, i.e., drawing the same conclusion from two causes, is troublesome: how to determine whether the same reason can be deduced? Must it be possible to derive exactly the same conclusion from a smaller case like in Figure A.3? As uncontrollable transitions cannot be influenced by a modeler, can a cause therefore be minimized by simply removing all nodes related to uncontrollable transitions? This would, in essence, only remove information which the modeler cannot use. Intuitively, removing information about uncontrollable transitions does not change the decision space while the semantics of the cause clearly changes.

A.3 Termination

Related to issues that arise with minimality of causes, is the termination of an algorithm for the derivation of all causes. Even when only the reduced paths are considered, infinitely many causes may be derived for a system. Therefore, a termination criterium is needed based on the “addition” of new information. In other words, a way of comparing two causes and assessing whether the two causes provide the same cause is needed. As shown earlier, comparing causes is problematic and a solution must be found to tackle this and the previous challenge.

Appendix B

Optimally ordering statements in PLCs

As explained in Section 6.5, a fluctuating cycle time has influence on the outcome of the controller; when the controller takes longer to respond, a product may have traveled further than desired. Likewise, a wrong ordering of transitions may induce the same problems when no urgency loop is added to the program.

To illustrate this, consider the system of Figure B.1 for which two possible orderings of the code fragments can be thought of: either the statement for the edge labeled with σ, or the statement for the edge labeled τ can appear first. The two different orderings are shown in Figure B.2.

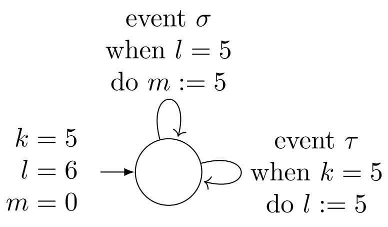


Figure B.1: A simple automaton.

As the reading and writing of the inputs and outputs takes up a certain amount of time, the wrong ordering may increase the response time of a PLC program. Assuming that all instructions take roughly the same amount of time, an exaggerated visual comparison is made between the two orderings of Figure B.2 in Figure B.3. In this figure, read and write are respectively used to denote the reading of the inputs and the writing of the outputs.

The ordering of statements must be sought in the dependencies between the variables, as data is the only concept remaining in the PLC. The question arises whether it is possible to allow for the reordering of code fragments, based on the dependencies between the variables, such that an optimal ordering is obtained, i.e., a program with

```
// event σ
if l = 5 then
    m := 5
end
// event τ
if k = 5 then
    l := 5
end
```

```
// event τ
if k = 5 then
    l := 5
end
// event σ
if l = 5 then
    m := 5
end
```

Figure B.2: Two different orderings of code fragments.

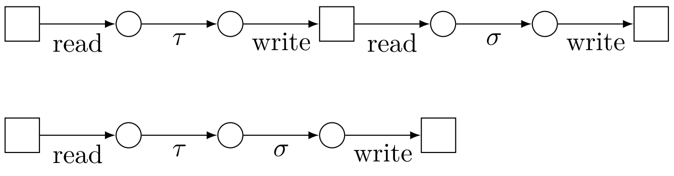


Figure B.3: Implications on the cycle time, due to ordering of transitions.

a minimal response time.

The problem of optimally ordering statements or scheduling tasks is well-known in computer science. Again Kahn's algorithm (Kahn, 1962) may be used to find an optimal ordering. For this particular application, a dependency graph must be created for the relations between the transitions; a transition with a guard $m = 5$ depends on all edges having an assignment $m := 5$. Note that such a strict dependency may not always be found, as it is not statically determinable whether a transition with a guard $m = 5$ depends on an edge with an assignment $m := m + 1$. Heuristic rules may be applied to link transitions till a satisfying result is obtain.

As a final remark, Kahn's algorithm can only be used for acyclic dependency graphs. The algorithm returns a list that contains an optimal ordering that does not necessarily need to be unique. Based on the resulting ordering of this algorithm, the code fragments can be placed in the code. If a cyclic dependency graph is obtained, each cycle may be reduced to a single node to obtain an acyclic graph. The cyclic dependencies may be substituted by a single node, as an optimal ordering can never be obtained. Hence, these transitions may be placed in arbitrary order before their dependent transitions.

Bibliography

N. Aizenbud-Reshef, B.T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006.

K. Åkesson, H. Flordal, and M. Fabian. Exploiting modularity for synthesis and verification of supervisors. *IFAC Proceedings Volumes*, 35(1):175–180, 2002.

R. Alur. Timed automata. In *International Conference on Computer Aided Verification*, pages 8–22. Springer, 1999.

D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL – a tool suite for automatic verification of real-time systems. In *Hybrid Systems III*, pages 232–243. Springer, 1996.

G. Black and V.V. Vyatkin. Intelligent component-based automation of baggage handling systems with iec 61499. *IEEE Transactions on Automation Science and Engineering*, 7(2):337–351, 2010.

U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M.S. Marshall. GraphML progress report structural layer proposal. In *International Symposium on Graph Drawing*, pages 501–512. Springer, 2001.

N.C.W.M. Braspenning. *Model-based integration and testing of high-tech multi-disciplinary systems.* PhD thesis, Eindhoven University of Technology, 2008.

J.L. Camus. SCADE: Implementation and Applications. In *Formal Methods: Industrial Use from Model to the Code*, pages 225–271. John Wiley & Sons, Inc., Hoboken, NJ, USA, March 2013.

C.G. Cassandras and S. Lafortune. *Introduction to discrete event systems.* Springer Science & Business Media, 2009.

J.P. Cavada, C.E. Cortés, and P.A. Rey. A simulation approach to modelling baggage handling systems at an international airport. *Simulation Modelling Practice and Theory*, 75:146–164, 2017.

G. Cengic, K. Akesson, B. Lennartson, C. Yuan, and P. Ferreira. Implementation of full synchronous composition using IEC 61499 function blocks. In *IEEE International Conference on Automation Science and Engineering, 2005.*, pages 267–272. IEEE, 2005.

M. Chechik and A. Gurfinkel. A framework for counterexample generation and exploration. *International Journal on Software Tools for Technology Transfer*, 9(5-6): 429–445, 2007.

K. Claessen, N. Een, M. Sheeran, N. Sörensson, A. Voronov, and K. Åkesson. SAT-solving in practice, with a tutorial example from supervisory control. *Discrete Event Dynamic Systems*, 19(4):495–524, 2009.

S. Cranen, B. Luttik, and T.A.C. Willemse. Evidence for fixpoint logic. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 41. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015. 24th Conference on Computer Science Logic.

R. De Neufville. The baggage system at Denver: prospects and lessons. *Journal of Air Transport Management*, 1(4):229–236, 1994.

A.H. Eden and T. Mens. Measuring software flexibility. *IEE Proceedings-Software*, 153(3):113–125, 2006.

R. Ehlers, S. Lafortune, S. Tripakis, and M. Vardi. Bridging the gap between supervisory control and reactive synthesis: case of full observation and centralized control. *IFAC Proceedings Volumes*, 47(2):222–227, 2014.

Esterel Technologies. SCADE Suite. `http://www.esterel-technologies.com/products/scade-suite`, 2017.

Z. Fei, S. Miremadi, K. Åkesson, and B. Lennartson. Efficient symbolic supervisor synthesis for extended finite automata. *IEEE Transactions on Control Systems Technology*, 22(6):2368–2375, 2014.

S.T.J. Forschelen, J.M. van de Mortel-Fronczak, R. Su, and J.E. Rooda. Application of supervisory control theory to theme park vehicles. *Discrete Event Dynamic Systems*, 22(4):511–540, 2012.

O.C.Z. Gotel and C.W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering*, pages 94–101. IEEE, 1994. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=292398`.

L. Grigorov, B.E. Butler, J.E.R. Cury, and K. Rudie. Conceptual design of discrete-event systems using templates. *Discrete Event Dynamic Systems*, 21(2):257–303, 2011.

J.F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The formal specification language mCRL2. In *Dagstuhl Seminar Proceedings.* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

M. Haoues, A. Sellami, H. Ben-Abdallah, and L. Cheikhi. A guideline for software architecture selection based on ISO 25010 quality related characteristics. *International Journal of System Assurance Engineering and Management*, pages 1–24, 2016.

D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. ISSN 0167-6423.

M.T. Heath. *Scientific computing*. McGraw-Hill New York, 2002.

A. Heegren, M. Fabian, and B. Lennartson. Synchronised execution of discrete event models using sequential function charts. In *Decision and Control, 1999. Proceedings of the 38th IEEE Conference on*, volume 3, pages 2237–2242. IEEE, 1999.

T.A. Henzinger. The theory of hybrid automata. In *Verification of Digital and Hybrid Systems*, pages 265–292. Springer, 2000.

IBM Rational DOORS. URL `https://www.ibm.com/us-en/marketplace/rational-doors`. Last accessed on the 27th of December 2017.

IEEE. Systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, Dec 2010.

IEEE. Ieee standard for software quality assurance processes. *IEEE Std 730-2014 (Revision of IEEE Std 730-2002)*, pages 1–138, June 2014.

D. Jackson. Dependable software by design. *Scientific American*, 294:69–75, 2006.

S. Jiang and R. Kumar. Supervisory control of discrete event systems with CTL* temporal logic specifications. *SIAM Journal on Control and Optimization*, 44(6): 2079–2103, 2006.

M. Johnstone, D. Creighton, and S. Nahavandi. Enabling industrial scale simulation/emulation models. In *Proceedings of the 39th Winter Simulation Conference*, pages 1028–1034. IEEE Press, 2007.

M. Johnstone, D. Creighton, and S. Nahavandi. Simulation-based baggage handling system merge analysis. *Simulation Modelling Practice and Theory*, 53:45–59, 2015.

A.B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5 (11):558–562, 1962.

R.M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

D. Luenberger. An introduction to observers. *IEEE Transactions on automatic control*, 16(6):596–602, 1971.

C. Ma and W.M. Wonham. Nonblocking supervisory control of state tree structures. *IEEE Transactions on Automatic Control*, 51(5):782–793, 2006.

R. Malik and H. Flordal. Yet another approach to compositional synthesis of discrete event systems. In *9th International Workshop on Discrete Event Systems, 2008*, pages 16–21. IEEE, 2008.

J. Markovski, K.G.M. Jacobs, D.A. van Beek, L.J. Somers, and J.E. Rooda. Coordination of resources using generalized state-based requirements. In *WODES*, pages 287–292, 2010a.

J. Markovski, D.A. van Beek, R.J.M. Theunissen, K.G.M. Jacobs, and J.E. Rooda. A state-based framework for supervisory control synthesis and verification. In *49th IEEE Conference on Decision and Control (CDC)*, pages 3481–3486. IEEE, 2010b.

MathWorks. MATLAB Simulink. `https://www.mathworks.com/products/simulink.html`. Accessed on 16th of January 2017.

I. McGregor. The relationship between simulation and emulation. In *Proceedings of the 34th Winter Simulation Conference*, volume 2, pages 1683–1688. IEEE, 2002.

T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

S. Miremadi and B. Lennartson. Symbolic on-the-fly synthesis in supervisory control theory. *IEEE Transactions on Control Systems Technology*, 24(5):1705–1716, 2016.

S. Mohajerani, R. Malik, and M. Fabian. A framework for compositional synthesis of modular nonblocking supervisors. *IEEE Transactions on Automatic Control*, 59 (1):150–162, 2014.

M.R. Mousavi, M.A. Reniers, and J.F. Groote. Notions of bisimulation and congruence formats for SOS with data. *Information and Computation*, 200(1):107–147, 2005.

M.R. Mousavi, M.A. Reniers, and J.F. Groote. SOS formats and meta-theory: 20 years after. *Theoretical Computer Science*, 373(3):238–272, 2007.

D.E. Nadales Agut, D.A. van Beek, and J.E. Rooda. Syntax and semantics of the compositional interchange format for hybrid systems. *The Journal of Logic and Algebraic Programming*, 82(1):1–52, 2013.

E. Nuutila and E. Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, 1994.

R. Nystrom. *Game programming patterns*. Genever Benning, 2014.

L. Ouedraogo, R. Kumar, R. Malik, and K. Åkesson. Symbolic approach to nonblocking and safe control of extended finite automata. In *Automation Science and Engineering (CASE), 2010 IEEE Conference on*, pages 471–476. IEEE, 2010.

M.H. Park and M. Kim. A distributed synchronization scheme for fair multi-process handshakes. *Information Processing Letters*, 34(3):131–138, 1990.

R.S. Pressman. *Software engineering: a practitioner's approach.* Palgrave Macmillan, 2005.

P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM journal on control and optimization*, 25(1):206–230, 1987.

F.F.H. Reijnen, M.A. Goorden, J.M. van de Mortel-Fronczak, and J.E. Rooda. Supervisory control synthesis for a waterway lock. In *Control Technology and Applications (CCTA), 2017 IEEE Conference on*, pages 1562–1563. IEEE, 2017.

J.C. Rijsenbrij and J.A. Ottjes. New developments in airport baggage handling systems. *Transportation Planning and Technology*, 30(4):417–430, 2007.

S. Russo. Finding a way in the model driven jungle: Invited keynote talk. In *ACM International Conference Proceeding Series*, pages 13–15. Universita degli Studi di Napoli Federico II, Naples, Italy, February 2016.

Stateflow. Mathworks. URL `https://www.mathworks.com/products/stateflow.html`. Last accessed on the 27th of December 2017.

L. Swartjes, R. Su, and J.E. Rooda. A case study on timed supervisory control on a linear cluster tool using aggregated timed synthesis. In *Control and Automation (ICCA), 2011 9th IEEE International Conference on*, pages 1189–1194. IEEE, 2011.

L. Swartjes, D.A. van Beek, and M.A. Reniers. Towards the removal of synchronous behavior of events in automata. *IFAC Proceedings Volumes*, 47(2):188–194, 2014. 12th International Workshop on Discrete Event Systems (WODES).

L. Swartjes, M.A. Reniers, D.A. van Beek, and W.J. Fokkink. Why is my supervisor empty? Finding causes for the unreachability of states in synthesized supervisors. In *13th International Workshop on Discrete Event Systems (WODES)*, pages 14–21. IEEE, 2016.

L. Swartjes, L.F.P. Etman, J.M. van de Mortel-Fronczak, J.E. Rooda, and L.J.A.M. Somers. Simultaneous analysis and design based optimization for paper path and timing design of a high-volume printer. *Mechatronics*, 41:82–89, 2017a.

L. Swartjes, D.A. van Beek, W.J. Fokkink, and J.A.W.M. van Eekelen. Model-based design of supervisory controllers for baggage handling systems. *Simulation Modelling Practice and Theory*, 78:28–50, 2017b.

G. Tassey. Standardization in technology-based markets. *Research policy*, 29(4): 587–602, 2000.

M.H. ter Beek, M.A. Reniers, and E.P. de Vink. Supervisory controller synthesis for product lines using CIF 3. In *International Symposium on Leveraging Applications of Formal Methods*, pages 856–873. Springer, 2016.

R.J.M. Theunissen, M. Petreczky, R.R.H. Schiffelers, D.A. van Beek, and J.E. Rooda. Application of supervisory control synthesis to a patient support table of a magnetic resonance imaging scanner. *IEEE Transactions on Automation Science and Engineering*, pages 20–32, 2014.

A. Vahidi, M. Fabian, and B. Lennartson. Efficient supervisory synthesis of large systems. *Control Engineering Practice*, 14(10):1157–1167, 2006.

D.A. van Beek, Pieter J.L. Cuijpers, Jasen Markovski, D.E. Nadales Agut, and J.E. Rooda. Reconciling urgency and variable abstraction in a hybrid compositional setting. In *FORMATS*, pages 47–61. Springer, 2010.

D.A. van Beek, W.J. Fokkink, D. Hendriks, A. Hofkamp, J. Markovski, J.M. van de Mortel-Fronczak, and M.A. Reniers. CIF 3: model-based engineering of supervisory controllers. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 575–580. Springer, 2014.

B. van der Sanden, M.A. Reniers, M. Geilen, T. Basten, J. Jacobs, J. Voeten, and R. Schiffelers. Modular model-based supervisory controller design for wafer logistics in lithography machines. In *Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on*, pages 416–425. IEEE, 2015.

M. van Weerdenburg and M.A. Reniers. Structural operational semantics with first-order logic. *Electronic Notes in Theoretical Computer Science*, 229(4):85–106, 2009.

M. von der Beeck. A comparison of statecharts variants. In Hans Langmaack, Willem P. de Roever, and Jan Vytopil, editors, *Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 128–148, Lübeck, Germany, 1994. Springer.

V. Vyatkin. Software engineering in industrial automation: State-of-the-art review. *IEEE Transactions on Industrial Informatics*, 9(3):1234–1249, 2013.

S. Wagner. *Software product quality control*. Springer, 2013.

J. Whittle, J. Hutchinson, and M. Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE software*, 31(3):79–85, 2014.

S. Winkler and J. von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software & Systems Modeling*, 9(4):529–565, September 2010. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-009-0145-0. URL `http://link.springer.com/10.1007/s10270-009-0145-0`.

W.M. Wonham, K. Cai, and K. Rudie. Supervisory control of discrete-event systems: A brief history–1980-2015. *IFAC-PapersOnLine*, 50(1):1791–1797, 2017.

S. Woods and Q. Yang. The program understanding problem: analysis and a heuristic approach. In *Proceedings of the 18th international conference on Software engineering*, pages 6–15. IEEE Computer Society, 1996.

J. Yan and V. Vyatkin. Distributed execution and cyber-physical design of baggage handling automation with iec 61499. In *9th IEEE International Conference on Industrial Informatics (INDIN)*, pages 573–578. IEEE, 2011.

yWorks. yEd graph editor: high-quality diagrams made easy. URL `https://www.yworks.com/products/yed`. Last accessed on the 6th of January 2017.

Curriculum vitae

Lennart Swartjes was born on the 14th of December 1987 in Utrecht, The Netherlands. After finishing Gymnasium in 2006 at Oosterlicht College in Nieuwegein he studied Mechanical Engineering at Eindhoven University of Technology in Eindhoven. In 2012 he graduated in the former Manufacturing Networks group on the paper path layout and timing design of high-volume printers. From February 2013 he performed research as a Ph.D. student at Eindhoven University of Technology at Eindhoven. The results of this research are presented in this dissertation. Since 2017 he is employed at Vanderlande Industries as a model-based design engineer.

Societal summary

A baggage handling system (BHS) is a key component of an airport. A BHS is responsible for transporting baggage items from designated entry points, such as check-ins, to designated exits, such as airplanes. To this end, a BHS consists of many kilometers of conveyor belts and specialized components for the routing of baggage items. To control these systems, a supervisory controller is used. Such a controller sends control actions to the actuators, e.g., motors, based on information obtained by sensors, e.g., photo cells.

Due to various reasons, the design of these supervisory controllers is difficult and lengthy. Using a model-based design approach with early integration may reduce these difficulties and lengthy design cycles. The key idea of this approach is that the (controlled) behavior of the system is captured in a model, such that it can be tested before the actual controller is coded and before the actual system is built.

In this thesis, it is investigated how these models can be built in such a way that the quality and flexibility is increased while the complexity is simultaneously decreased. Methods are introduced to aid the end-user in a process called supervisory controller synthesis; a process in which the controller is automatically synthesized based on a model of the system requirements. Moreover, the traceability between models is investigated when performing model transformations. Finally, the implementation on a real-time platform is investigated and the feasibility of this implementation is shown by performing three industrial-sized cases.

All in all, this thesis provides modelling considerations while also adding contributions to the usability of model-based design for large industrial systems such as BHSs.

Titles in the IPA Dissertation Series since 2015

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems*. Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming*. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10