TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Echtzeitsysteme und Robotik

Game-based Synthesis for Distributed Control of Industrial Assembly Lines

Michael Stefan Geisinger

Vollständiger Abdruck der von der Fakultät für Informatik der

Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Hans Michael Gerndt

Prüfer der Dissertation:

1. Univ.-Prof. Dr.-Ing. habil. Alois Knoll
2. Prof. George J. Pappas, Ph.D.
   University of Pennsylvania, Philadelphia/USA

Die Dissertation wurde am 27.01.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 15.06.2015 angenommen.

Abstract

The manufacturing industry nowadays uses a high degree of automation that contributes significantly to its commercial success. Set-up of an automation system is usually very expensive and only pays off by amortization of costs over its life span. Shorter product life cycles and increasing personalization of products, which are the results of the so-called fourth industrial revolution, endanger this concept: it becomes more and more important that not only set-up, but also adaptation of automation systems to changing production requirements is cost-efficient. Modification of the control software plays an important role in this context, because programming of such systems often takes place at a low level and is hence restricted to experts who are aware of the consequences of their design decisions.

This work presents a model-driven workflow that aims at reducing the costs of adapting the control software in industrial automation systems. Its goal is to replace the tedious and error-prone manual adaptation by an automatic approach which is based on synthesis. This has two major benefits: first, the possibility for non-experts to adapt the control software allows experts in industrial automation to focus on other tasks, reducing overall costs. Second, programming errors are avoided, because re-synthesis is triggered with the push of a button and the synthesized control programs are correct by construction.

In order to achieve these benefits, this work introduces formalisms for describing automation systems and the production tasks to accomplish. The sequence of primitive control actions to execute is then automatically synthesized by a solver that is based on game theory. The result is transformed into executable code, which allows both offline simulation as well as online execution within the real automation system. For distributed control systems, the approach is capable of generating multiple control programs that cooperate in order to achieve the production goal. The presented workflow integrates nicely with existing concepts from industrial automation, such as programmable logic controller (PLC) programming languages as defined in IEC 61131-3.

The contributions of this work are the definition of a suitable domain specific modeling language for precise description of automation systems and automation tasks, the transformation of the model into an input for the game-based solver as well as a software library for simulation and execution of the generated control programs on centralized and distributed target control units.

In order to show the operability of the approach, the development workflow is applied to various industrial-grade automation systems. Programmable logic controllers, personal computers and microcontrollers are used as target platforms. For the scenarios examined in this work, synthesis time is in the range of seconds to minutes and hence acceptable in the target domain. Evaluation also includes experiments with cost-based optimization and parallel execution of individual production steps.

Zusammenfassung

In der produzierenden Industrie wird heute ein hoher Grad an Automatisierung eingesetzt, der maßgeblich zum wirtschaftlichen Erfolg beiträgt. Der Aufbau eines Automatisierungssystems ist üblicherweise sehr teuer und zahlt sich nur durch die Amortisierung der Kosten über die Laufzeit aus. Kürzere Produktlebenszyklen und zunehmende Individualisierung von Produkten, die eine Folge der so genannten vierten industriellen Revolution sind, gefährden dieses Konzept: Es wird immer wichtiger, dass Automatisierungssysteme nicht nur kosteneffizient aufgebaut, sondern bei veränderten Produktionsbedingungen auch effizient umgerüstet werden können. Die Anpassung der Steuerungssoftware spielt hierbei eine große Rolle, da die Programmierung oft hardwarenah erfolgt und damit Experten vorbehalten ist, die sich der Wechselwirkungen ihrer Design-Entscheidungen bewusst sind.

Diese Arbeit stellt einen modellgetriebenen Ansatz zur Senkung der Kosten bei der Anpassung der Steuerungssoftware in Automatisierungssystemen vor. Dabei wird der mühsame und fehlerbehaftete manuelle Anpassungsprozess durch einen Synthese-Ansatz ersetzt. Dies hat zwei Vorteile: Einerseits ergeben sich Kostensenkungen, da sich Automatisierungs-Experten auf andere Aufgaben konzentrieren können. Andererseits werden Programmierfehler ausgeschlossen, da eine erneute Synthese auf Knopfdruck möglich ist und das Ergebnis konstruktionsbedingt korrekt ist.

Um diese Ziele zu erreichen, führt diese Arbeit eine Sprache ein, mit der Automatisierungssysteme und die damit durchzuführenden Produktionsaufgaben formal beschrieben werden. Die konkreten Steuerbefehle werden dann mittels eines auf der Spieltheorie basierenden Lösungsalgorithmus synthetisiert. Der resultierende ausführbare Code erlaubt Offline-Simulation und Online-Ausführung auf der Zielplattform. In verteilten Systemen werden mehrere Steuerungsprogramme generiert, die gemeinsam die Produktionsaufgabe lösen. Der vorgestellte Ansatz integriert sich gut in existierende Konzepte der Automatisierungstechnik, wie z.B. Programmiersprachen nach IEC 61131-3.

Der Beitrag dieser Arbeit liegt in der Definition einer domänenspezifischen Modellierungssprache zur präzisen Beschreibung von Automatisierungssystemen und -aufgaben, der Transformation der Modelle in eine Eingabe für den Löser sowie einer Softwarebibliothek zur Simulation und Ausführung der generierten Steuerungsprogramme auf der (verteilten) Zielhardware.

Um die Funktionsfähigkeit des Ansatzes zu demonstrieren, wird er auf mehrere Automatisierungssysteme von industrieller Qualität angewendet. Dabei kommen speicherprogrammierbare Steuerungen, Personal Computer und Mikrocontroller als Zielplattformen zum Einsatz. Für die in dieser Arbeit untersuchten Szenarien liegen die Synthesezeiten im Bereich von Sekunden bis Minuten und sind damit akzeptabel für die Zieldomäne. Die Evaluierung umfasst auch Szenarien mit kostenbasierter Optimierung und paralleler Ausführung mehrerer einzelner Produktionsschritte.

Acknowledgments

First of all, I would like to thank my supervisor Prof. Dr.-Ing. Alois Knoll for the opportunity to prepare this thesis. His ideas, criticism and guidance helped to set the right focus for this work. I would also like to thank Prof. George J. Pappas for accepting to be my external advisor and for the interesting discussions on the similarities between his research and my work.

Many thanks go to my colleagues and former colleagues at the fortiss research institute and the Chair of Robotics and Embedded Systems. First and foremost, thanks to Dr. Harald Ruess for supporting my research in the course of my employment at fortiss. A big thanks to Dr. Christian Buckl for the intensive feedback, guidance and constructive criticism throughout the preparation of this work. Thanks to Dr. Chih-Hong Cheng for introducing me to the domain of software synthesis and for providing the GAVS+ solver that this work is based on. Thanks to Gerd Kainz for the joint effort to assemble and enhance the Festo MPS demonstrators used in this work. Finally, many thanks to all other colleagues and former colleagues for teaching me all the small things that are necessary to successfully complete such a work.

I would also like to thank my parents Karin and Franz for the superb education and support as well as lasting patience and faith in me finishing this work. Your frequent reminder to get this thesis done gave me the extra motivation to make it possible.

Last but not least, this work would not have been possible without the countless hours of work that have been put into free software and open source software by contributors all around the world. In order to give something back, the software written in the context of this work is available as open source and licensed under the GNU General Public License (GPL), Version 3.0.[a]

[a] Due to the use of software libraries under GPL (most notably GAVS+), this is the most liberal licensing scheme that is possible.

Contents

List of Acronyms **XI**

List of Symbols **XIII**

List of Figures **XVII**

1. Introduction **1**
1.1. Setting and Motivation 2
1.2. Goals of this Thesis 6
1.3. Main Contributions of this Thesis 8
1.4. Structure of this Thesis 9

2. Background and Trends **11**
2.1. Industrial Automation 12
2.2. Modeling and Model-driven Development 19
2.3. Games and Game-based Synthesis 25

3. Overview of the Approach **31**
3.1. User Roles and Domain Knowledge 32
3.2. Derived Workflow 33
3.3. Scope of this Work 37
3.4. Introduction of the Running Example 39

4. System Modeling and Task Description **43**
4.1. Modeling Overview 44
4.2. Formal Description of Modular Assembly Lines 44
4.3. Formal Description of Automation Tasks 53
4.4. Discussion and Application to Running Example 56
4.5. Summary 70
4.6. Related Work 71

5. Industrial Control Program Generation Workflow **73**
5.1. Approach . . . 74
5.2. Constraint Checking . . . 77
5.3. Model-to-model Transformation . . . 79
5.4. Model-to-text Transformation . . . 86
5.5. Game-based Solving . . . 93
5.6. Translation of Solver Output to Control Programs . . . 96
5.7. Discussion and Application to Running Example . . . 100
5.8. Summary . . . 102
5.9. Related Work . . . 103

6. Platform Mapping and Execution **105**
6.1. Platform Mapping Overview . . . 106
6.2. Generation of Platform Mapping Code . . . 106
6.3. Mapping of Behavioral Primitives . . . 107
6.4. Manually Written Platform Library . . . 111
6.5. Discussion and Application to Running Example . . . 113
6.6. Summary . . . 114

7. Realization and Evaluation **115**
7.1. Model-driven Development Tool MGSyn . . . 116
7.2. Simulation of Control Program Execution . . . 118
7.3. Evaluation Overview . . . 119
7.4. Evaluation of the Running Example . . . 121
7.5. Evaluation of Circular Material Flow Example: Focus on Parallel Execution . . . 122
7.6. Evaluation of Bidirectional Material Flow Example: Focus on Decentralized Execution . . . 126
7.7. Summary . . . 132

8. Conclusion **135**

A. Model Transformation: Token-based Ownership of Predicates **139**

Index **143**

Bibliography **145**

List of Acronyms

API	application programming interface
ASCII	American Standard Code for Information Interchange (encoding scheme)
BDD	binary decision diagram
BNF	Backus-Naur form
CNC	computerized numerical control
DCS	distributed control system
DML	dedicated manufacturing line
DSL	domain-specific language
DSM	domain-specific modeling
ECU	electronic control unit
EMF	Eclipse Modeling Framework
EMP	Eclipse Modeling Project
ERP	enterprise resource planning
ET	execution time
FBD	function block diagram
FMS	flexible manufacturing system
FSM	finite state machine
GAVS	Game Arena Visualization and Synthesis (software framework)
GAVS+	Game Arena Visualization and Synthesis Plus! (software framework)
GMF	Graphical Modeling Framework (Eclipse platform framework)
GUI	graphical user interface
IDE	integrated development environment
IEC	International Electrotechnical Commission (standardization organization)
IL	instruction list
I/O	input/output
IP	Internet Protocol
IPC	industrial personal computer

IT information technology
LD ladder diagram
LTL linear temporal logic
M2M model-to-model transformation
M2T model-to-text transformation
MAL modular assembly line
MBD model-based design/development
MDD model-driven development
MDE model-driven engineering
MES manufacturing execution system
MGSyn Model, Game and Synthesis (game-based synthesis approach)
MOF Meta-Object Facility
MPI Multi-Point Interface (Siemens SIMATIC S7 PLC interface)
MPS Modular Production System (product line from Festo)
OCL Object Constraint Language
OLE Object Linking and Embedding (object system and protocol)
OMG Object Management Group
OPC open platform communications (originally: OLE for process control)
OS operating system
PC personal computer
PCF primitive control function
PDDL Planning Domain Definition Language
PID proportional-integral-derivative (controller type)
PLC programmable logic controller
POU program organization unit
PWM pulse-width modulation
RAM random access memory
RMS reconfigurable manufacturing system
SCADA supervisory control and data acquisition
SFC sequential function chart
SME small and medium enterprise
ST structured text
TCP Transmission Control Protocol
UDP User Datagram Protocol
UML Unified Modeling Language
UML2 Unified Modeling Language 2
WC worst-case
WCET worst-case execution time
XMI XML Metadata Interchange
XML Extensible Markup Language

List of Symbols

Symbol	Explanation
$\odot$	Sequential composition operator
$\otimes$	Parallel composition operator
$\rightarrow$	Sequential composition
$\|\|$	Parallel composition
A	List of arguments
a	Argument
a	Number of arguments in list
$\mathcal{A}$	Arena
$\mathcal{B}$	Set of behavioral interfaces
b	Behavioral interface
b	Number of behavioral interfaces in set (if used as index)
β	Number of behavioral interfaces triggerings in trace
$\mathbb{B}$	Set of Boolean values
$\mathfrak{b}$	Barrier
$\mathcal{C}$	Set of conditions
c	Condition
$\mathcal{C}_\mathcal{I}$	Set of initial conditions
c_i	Initial condition
$\mathcal{C}_\mathcal{P}$	Set of preconditions
c_p	Precondition
$\mathfrak{C}$	Set of check rules
$\mathfrak{c}$	Check rule
$\mathfrak{c}$	Number of check rules (if used as index)
d	Degree of parallelization
δ	Delay
$\mathcal{E}$	Set of effects
e	Effect

Continued on next page

Symbol	Explanation
$\widehat{\mathcal{E}}$	Set of conditional effects
$\hat{e}$	Conditional effect
E	Edge relation
e	Event (if used in context of time modeling)
ε	Accuracy
ϵ	Empty string
η	Cost
$\eta_{\max}$	Cost bound
g	Goal specification
$\mathcal{G}$	Game
$\mathfrak{H}$	Hardware model
$\mathcal{I}$	Set of input signals
ι	Input signal
ι	Number of input signals in set (if used as index)
$\vec{\iota}$	Input signal assignment function
j	Jitter
L	List of overlapping positions
ℓ	Overlapping position
$\mathcal{M}$	Set of all process models
$\mathfrak{M}$	Process model
$\mathcal{O}$	Set of output signals
o	Output signal
o	Number of output signals in set (if used as index)
$\vec{o}$	Output signal assignment function
φ	Module type
Φ	Set of module types
$\mathfrak{P}$	Plant model
$\mathcal{P}$	Set of configurable parameters
p	Configurable parameter
p	Number of configurable parameters in set (if used as index)
$\vec{p}$	Parameter assignment function
$\hat{p}$	Primitive control function
$\ddot{p}$	Parallelizable flag
Π	Set of operating positions
π	Operating position
π	Play (if used in context of game theory)
P	Time-penalty function
Ψ	Set of module instances
ψ	Module instance
$\mathcal{R}$	Set of sensor result conditions
r	Sensor result condition
ρ	Sensor response value

Continued on next page

Symbol	Explanation
RW	Set of reading and writing ECU instances
s	Specification
$\mathcal{S}$	Set of potential predicate valuations
σ	Solver type
Σ	Alphabet (ASCII character set)
$\mathfrak{T}$	Task model
$\mathcal{T}$	Set of operating position types
t	Operating position type
$\mathtt{t}$	Number of object types (if used as index)
$\mathtt{t}_v$	Allowed object types function for predicate v
t	Logical time (if used in context of time modeling)
τ	Physical time
Θ	Set of operating position instances
θ	Operating position instance
$\vec{\theta}$	Operating position mapping function
$\mathcal{U}$	Set of ECUs
u	ECU
$\widehat{\mathcal{U}}$	Set of ECU instances
$\hat{u}$	ECU instance
$\vec{u}$	ECU instance parameter assignment function
$\mathcal{V}$	Set of predicates (variables)
v	Predicate (variable)
$\mathcal{V}$	Set of vertices (if used in context of game theory)
v	Vertex (if used in context of game theory)
$\mathcal{X}$	Set of objects
x	Object
x	Number of objects in set (if used as index)
x	Payload (if used in context of time modeling)
Ξ	Set of platform types
ξ	Platform type
$\mathcal{W}$	Set of work pieces
w	Work piece
$\mathcal{W}$	Winning set (if used in context of game theory)

List of Figures

1.1. Comparison of traditional and proposed programming workflow 7
1.2. Graphical representation of the structure of this thesis 9

2.1. Paradigm shifts in manufacturing in the last two centuries 12
2.2. Example of an automation pyramid 14
2.3. PLC programming language examples 17
2.4. Computer programming and modeling languages abstraction 21
2.5. Metamodeling concept with concrete example 22
2.6. Example for game-based synthesis 28

3.1. User roles and domain knowledge in control software design 32
3.2. Traditional bottom-up control program design 35
3.3. Proposed confluent workflow for control program synthesis 35
3.4. Running example 39

4.1. Generalized representation of components from the running example . . 57

5.1. Comparison of centralized and decentralized control strategies 75

6.1. Inter-ECU networking class hierarchy 111
6.2. ECU class hierarchy 112
6.3. Communication class hierarchy 112
6.4. Primitive control function class hierarchy 113

7.1. MGSyn main application window 116
7.2. MGSyn configuration dialog 117
7.3. Simulating execution of decentralized control programs for four ECUs . 119
7.4. Circular material flow example 123
7.5. Photos of real plant implementing circular material flow 123
7.6. Bidirectional material flow example 128
7.7. Photos of real plant implementing bidirectional material flow 128

CHAPTER 1

Introduction

Contents

1.1. Setting and Motivation 2
1.2. Goals of this Thesis 6
1.3. Main Contributions of this Thesis 8
1.4. Structure of this Thesis 9

Overview

Automation systems are very common today in industrial branches that involve medium or high-volume production. Without industrial automation, many products that are part of our everyday life would not be as ubiquitous, high-quality and cheap. Automation systems keep the product quality at a constant high level while increasing volume of production significantly.

However, setup and maintenance of automation systems and the software used to control them poses significant challenges of both financial and technical nature. Coping with these challenges is a prerequisite for economic success and hence failing to do so is a severe risk for a manufacturing company.

This chapter first depicts which *technical* challenges and risks are present in design of industrial automation systems and their evolution over time. It then motivates the need for partial automation of the design process of control software for a specific class of industrial automation systems in order to cope with the presented challenges and to mitigate some of the risks.

1.1. Setting and Motivation

1.1.1. Advantages of Industrial Automation

The term *automatic* "pertains to a process or equipment that, under specified conditions, functions without human intervention" [IEC06]. An *automation system* is a system that "employs means to enable self-acting functions" [IEC06]. *Industrial automation* is the application of automation in industrial scope, i.e., to produce or process goods. In facilities with a high *degree of automation*, which is defined as the "proportion of automatic functions to the entire set of functions of a system or plant" [IEC06], production processes or processing of goods are typically carried out on *assembly lines*.

The major motivations for (industrial) automation as opposed to manual production and processing are to accommodate growth [BH07], "to improve accuracy, increase speed of service and output rate, to reduce labour cost and to improve service availability" [DJ91]. Although automation is often seen as a replacement for human workers, development of the past decades has shown that humans "are still very much involved in the monitoring and control of manufacturing operations, but our role has changed over the past decades. Rather than moving materials and parts from point A to point B, we now move ideas", said Jane Gerold, editorial director of Automation World magazine in 2004 [Ger04]. Where work places between humans and machines are shared, automation often extends "to functions that humans do not wish to perform", increasing the comfort and safety [PSW00].

1.1.2. Challenges of Industrial Automation

However, these advantages do not come for free and pose strict design requirements on the system, some of which are listed in the following.

Mastering Complexity

When compared to how easy it is for a human worker to achieve the same task, even trivial automation tasks result in "normally very complex" [BH07] automation systems that involve "a number of different systems that need to be designed and developed in parallel" [DF03]. This includes a number of *sensors* (for retrieving information from the environment), *actuators* (for controlled interaction with the environment) as well as associated communication infrastructure and *control programs*. The fact that development of such systems is complex is underlined by the insight that especially in the early phases after introduction of an automation system, service levels might be adversely affected [NB04]. "This is often due to the need for substantial testing, commissioning and 'snagging' (i.e., the rectification of faults) of automated equipment" [BH07].

The typical countermeasure is to employ development methodologies that reduce the complexity by **raising the level of abstraction**, for example model-based or model-driven development approaches [SV06]. Those approaches define clear interfaces between the individual crafts and enable control engineers to develop those systems in a more intuitive way, hence reducing potential for mistakes.

Decentralized Control

Many automation systems consist of multiple physically distributed *control units* or *control systems* that have to collaborate in order to achieve a global task [KKK84]. Coordination between control units in a plant takes place via shared communication media, which are called *fieldbuses* [Tho05]. Fieldbuses link *electronic control units (ECUs)* such as *programmable logic controllers (PLCs)* or *industrial personal computers (IPCs)* to the sensors and actuators in the plant. While sensors and actuators have been directly linked to PLCs and IPCs in traditional automation systems – often using analog signal transmission – the usage of (digital) bus systems has recently both reduced the length of the cables involved as well as the number of cables required, which "usually represents a significant part of the cost" [Tho05]. However, such advanced bus systems require adequate control software and configuration. For example, data exchange on real-time capable communication media requires according scheduling and prioritization [Tho05].

It is beneficial to represent the distributed nature of a control system in an appropriate model in order to be able to **reason about how the parts of a distributed application cooperate** and to **generate communication patterns** for exchanging information between control units.

Proof of Correctness

The implementation of the control programs for an automation system typically has to follow a given specification that includes, among others, safety requirements. Safety critical systems often need to be certified [FP04]. Certification is prevalent in industrial automation domains where interaction with humans takes place, such as a work cell shared between human workers and an industrial robot.

When certified automation systems are adapted, re-certification is typically required. This means that the adaptation of safety critical industrial control programs, for example due to changing production requirements or the rectification of faults, is very expensive. Synthesis of *correct by construction* control programs is hence desirable in order to reduce these costs. In the context of this work, *synthesis* means that a suitable algorithm automatically derives a control program that satisfies a given specification. Although the generated system still requires (re-)certification, such a process makes the development and adaptation of control software more cost-efficient.

Representation of Extra-functional Requirements

Many automation systems are part of time critical tasks. Be it that multiple actuators share the same working space or that a work piece requires processing within a fixed time bound because of the physical attributes of the process: in both cases, real-time requirements play an important role.

Many modeling techniques for representation of real-time requirements and respective realizations exist (e.g., [OMG11d, OMG07, SK00]). However, not all automation system designs treat real-time requirements as first class entities. In many cases, time-critical

automation tasks run on computing hardware (e.g., PLCs or IPCs) that an engineer selects according to an overestimation of the resources required. Evaluation of system timing typically boils down to empirical testing whether the system meets the requirements in all specified cases.

Among others, this approach bears the risk that an engineer who modifies an existing system does not properly understand how timing requirements have been realized, because they are implicitly "encoded" into the system. Hence, **explicit representation of extra-functional requirements** (such as timing aspects) in an appropriate language that allows automatic feasibility analysis should be the preferred way of modeling an automation system.

1.1.3. Overall Costs of Automation Systems

In traditional production scenarios, the high initial costs for installation and programming of an automation system amortize with the high volume of products manufactured. Although high-volume productions are still common, lot sizes become smaller in various industrial branches due to rapidly changing markets and increased customization of products [Fri11]. At the same time, *small and medium enterprises (SMEs)*, which have smaller lot sizes in general, start applying automation systems as well [SME09, SME12]. Automating a process only pays off if the costs required implementing and maintaining the automation system in relation to the volume of sales stay significantly below the costs of a comparable process with human interaction in the medium term or long run, depending on the business model.

Hence, significant costs not only originate from the installation of an automation system and its daily use, but also from the redesign and adaptation required when the production process needs to be changed. The respective time where no production is possible is called *changeover time* or *machine set-up time* as opposed to *machine uptime* [Fri11]. Boosting flexibility and adaptability of industrial automation lines in order to minimize changeover time is a key ingredient for overall cost-efficiency, because markets are nowadays highly volatile and demand is difficult to predict [CT02]. Facing unpredictable markets and short product life cycles, agility and flexibility is crucial for automation systems [BH07]. Fast response to changing market demands can decide about the market winner these markets [MJNT00].

The need to adapt an automation system is caused by, for example, altering an existing product or a new product line being developed. As products change, production lines also need to change their behavior and/or physical layout. Adaptations to industrial automation systems may consist of the following changes:

1. Adaptations to the physical production equipment as well as the *information technology (IT)* infrastructure used to control and maintain it (further called "adaptations to hardware"). Modular components and standardized interfaces typically reduce the costs for adaptations to hardware. For more details, see Section 2.1.1 and *flexible manufacturing systems (FMSs)*.
2. Adaptations to the *control programs* (i.e., the software that runs on a control unit in order to steer the process), configuration of communication and adaptations

to monitoring software (further called "adaptations to software"). The application of standards and suitable software engineering approaches typically reduce costs for adaptations to software. Such adaptations are manual tasks nowadays, although there is potential to automate them in many cases. However, there is still a need of research in this area. The concepts developed in this work address this need to some extent.

A change in the production process often requires both adaptations to hardware and software at the same time: on the one hand, adaptations to hardware are followed by adaptations in the software, because new sensors, actuators, control units or communication channels need to be integrated into the system. On the other hand, adaptations to the software often require the gathering of additional data and hence the installation of additional sensors or communication channels.

The vision of reconfigurable production lines is what is commonly referred to as *plug and produce* (an adaptation of the term *plug and play* from the consumer market), that is the software of an automation line reconfigures itself automatically after the hardware changes have been made.

1.1.4. Traditional Workflow: Bottom-up Development

The prevalent state-of-the-practice of software engineering for industrial automation systems is determined by current industrial standards such as IEC 61131-3 [IEC03a], IEC 61804 [IEC03b] and IEC 61499 [IEC12]. It is based on "painstakingly engineering sequences of relatively low-level control 'code' using standardized libraries of function blocks." [CGR+12b]. In order to control an actuator, a programmer typically deals with inputs and outputs at electrical signal level. Higher-level functionality, such as a control algorithm, bases on top of this manually written low-level software layer.

This traditional *bottom-up* style of programming leads to "inefficiencies in developing and maintaining industrial production control systems and it has negative impact on the quality and dependability of the control code itself" [CGR+12b]. One reason for this is that during development of the original control systems, developers implicitly encode requirements and design decisions into the control programs. Typical examples are the selection of an appropriate control strategy out of a set of possible implementations or the realization of timing requirements by selecting and "hard-coding" a scheduling policy for component execution. This approach is not suitable to allow inexperienced users, such as students or trainees, to modify existing automation systems without the risk of violating implicitly encoded design assumptions.

An effective workflow should provide means for experts in industrial automation to *explicitly* specify the assumed requirements in the initial design phase and allow modifications to the system at a high level of abstraction using views that hide the implementation details. Such modifications then do not necessarily have to be performed by experts, since violations to the assumptions are automatically detected and a solution respecting the explicit requirements is automatically derived.

1.1.5. Proposed Workflow: Confluent Development

This work proposes a new *confluent* style of programming the control software for industrial automation systems that includes both a top-down as well as a bottom-up workflow. This approach promises to mitigate some of the presented challenges and risks of the state-of-the-practice bottom-up development methodology.

The *top-down* part of the proposed workflow is based on describing *what* needs to be achieved by the production facilities instead of painstakingly encoding *how* to achieve certain production goals in sequences of low-level code. Developers use a formal language to specify the task to achieve. It explicitly lists the relevant design assumptions in form of quality-of-service contracts with respect to dependability requirements and efficiency constraints. This avoids implicit encoding of requirements and design decisions into the control software, shifting the task of writing control software from solution space to problem space. Finally, an automatic program synthesis algorithm generates the actual control program from the specification.

The *bottom-up* part of the workflow is concerned with providing an abstract model on which the refined outcome of the top-down part of the workflow is based.

An example of high-level instructions for an industrial automation task in informal form is "Drill and store red work pieces if they are oriented correctly." Given a formal model of the plant and its capabilities, the control software is automatically derived, provided the semantics of drilling, storing, work piece color and orientation are defined in a mathematically sound way.

To support the user in this workflow, a model-driven development tool is provided that implements the respective steps of the workflow. In case of adaptations to an automation system, the user adapts the plant model in the tool. Synthesis of a suitable new control program then happens without further user interaction.

The remainder of this chapter is composed as follows: the concrete goals of this work are summarized in Section 1.2, while Section 1.3 highlights the main contributions. Finally, a structural overview of this thesis is given in Section 1.4.

1.2. Goals of this Thesis

The five goals pursued in this work are:

1. **Programming of modular assembly lines with less or no expert knowledge.**

   Programming of industrial automation systems is a complex task that involves implementing *how* the system performs its duty. The idea behind this work is that, given a formal representation of *what* the system should do, an automatic process can synthesize the "how" from the "what", provided a formal model of the plant and its capabilities exists. This work focuses on a specific class of systems that we call *modular assembly lines (MALs)*.

   While the traditional development process of control software for industrial automation consists of manual implementation and adaptation phases as depicted in Figure 1.1 (a), this work splits the development process into three parts as il-

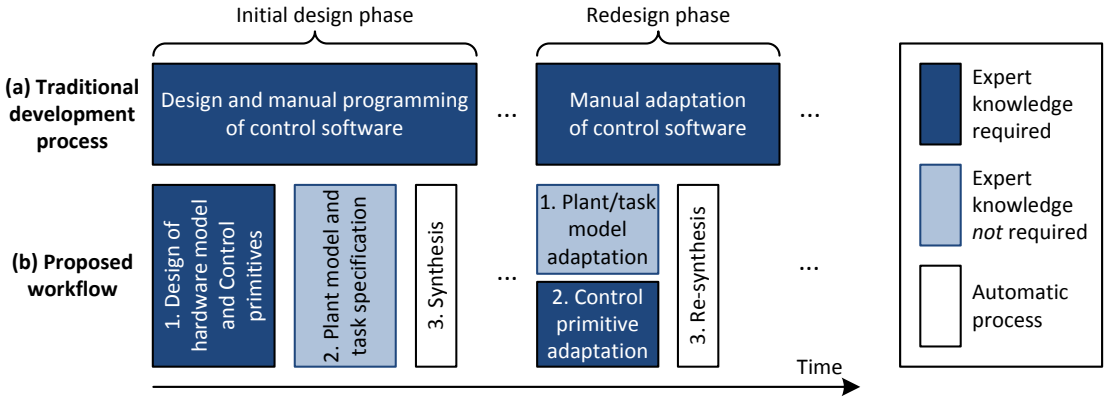


Figure 1.1.: Comparison of (a) traditional development process and (b) proposed workflow for developing control software in industrial automation. In the proposed workflow, specific expert knowledge is only required in a fraction of the development time due to the decomposition of the workflow into three parts. In addition, tasks in the redesign phase can be more easily parallelized.

lustrated in Figure 1.1 (b): first, experts in mechatronics and industrial automation model the basic components of the MAL under consideration and provide suitable *primitive control functions (PCFs)*. Those components are designed in a generic way and hence re-usable across applications. Second, users that do not necessarily have to have expert knowledge in industrial automation specify a high-level model of the concrete plant and the task to achieve. For this purpose, suitable tooling is provided in which the basic components written by the experts in mechatronics and industrial automation are available for use. Third, control software that satisfies the high-level specification is automatically synthesized.

2. **Cost-efficient adaptation of control software in automation systems.**

   Nowadays, production tasks are frequently adapted. Hence, it is important to ensure that the adaptation of a plant's control software is cost-efficient. Once the design process of the control software follows the proposed workflow, the software can be quickly adapted by altering the specification of the task to achieve and re-synthesizing the control program (compare Figure 1.1 (b)). For instance, the adaptation of the plant topology, altering of material flow, change of processing units and exchange of ECUs with different hardware is considered.

3. **Optimization of synthesized control software based on a notion of cost.**

   Multiple solutions exist for implementing a control program for a specific production goal on a given MAL. Selecting an optimized solution is important for maximum throughput and energy efficiency. In traditional automation design, control engineers chooses a meaningful realization or it is calculated by tools such as a *manufacturing execution system (MES)*. In this work, cost-based optimization is used to select a suitable solution.

4. **Synthesis of decentralized control software.**

   Industrial automation lines contain a number of ECUs that cooperate to achieve a global control task. Distributing a control algorithm over a number of ECUs is still done manually most of the time. The formal specification that forms the base of the approach presented in this work allows automatically deriving a suitable placement of parts of the control algorithm on the ECUs of the target system.

5. **Demonstration platform with heterogeneous control units.**

   To show the applicability of the approach in the real world, we chose a demonstration platform that has similar properties than systems used in practice. It employs PLCs, *personal computers (PCs)* and microcontrollers as decentralized control units.

1.3. Main Contributions of this Thesis

The three main contributions of this thesis are:

1. **Definition of a formal domain specific modeling language to describe MALs and respective automation tasks.**

   A precise, formal description is the base for reasoning about a system. This thesis introduces a formal description of MALs with the following features:

   - Description of processing modules with their so-called operating positions and behavioral interfaces (i.e., capabilities).
   - Description of module topology via overlapping operating positions.
   - Model of the discrete state space of the plant.
   - Properties of *work pieces* (i.e., the products being produced).
   - A selected set of properties of the ECUs in the automation system.

   The automation task to achieve is then formulated as a logical formula over the state space of the plant, given an initial state space assignment.

   Furthermore, this work also presents ideas on how extra-functional properties can be represented to enhance the expressiveness of the model.

   Details about the modeling of industrial automation systems and task description can be found in Chapter 4.

2. **Synthesis of decentralized control programs for MALs.**

   The formal task specification contains a goal condition that should eventually become true in order to achieve the desired automation task. The idea is to find a suitable sequence of control actions of processing modules in order to reach this goal.

   This work uses the solver *Game Arena Visualization and Synthesis Plus! (GAVS+)* [CKLB11] to achieve this task, which is based on game theory. For this purpose, the automation system model and the task specification are transformed into a suitable input language for the solver. Likewise, the result from invoking the solver, which is an imperative control program with guards and

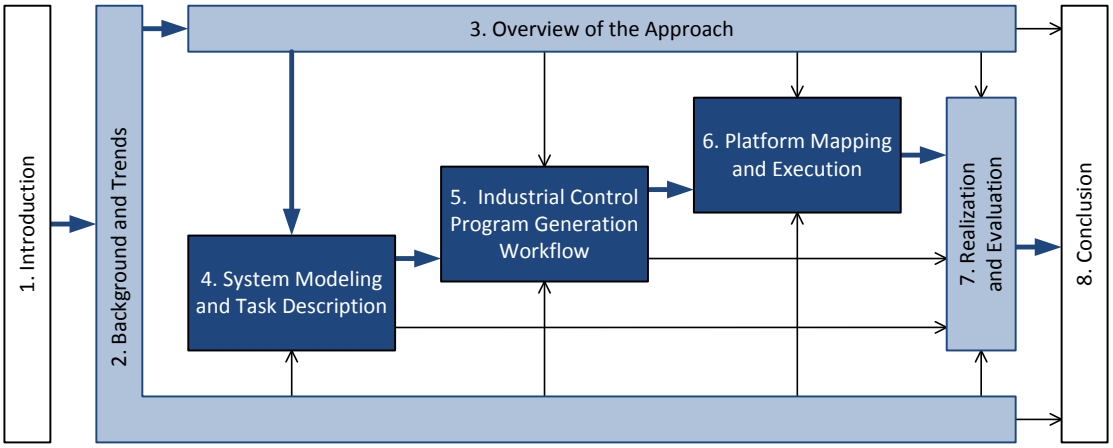


Figure 1.2.: Graphical representation of the structure of this thesis. All arrows indicate dependencies with regard to content (the tip of the arrow being the chapter that depends on the chapter where the arrow originates). Thick arrows indicate intended reading order.

explicit parallelism, is transformed into an executable program for simulation or execution on the real hardware.

Chapter 5 explains synthesis and the related processes in detail.

3. **Mapping of control programs to execution units for simulation and execution.**

   For the synthesized program to be executed on a target platform, "glue" code is required to implement the primitive actions described by the high-level behavioral interfaces in the model. Part of this code is automatically generated, but a thin layer of software corresponding to the driver layer in an *operating system (OS)* has to be manually written or designed, for instance the triggering of a control program on a PLC. Furthermore, depending on the target platform, abstraction or adequate static mapping of hardware resources and communication facilities (in case of a distributed control system) needs to be ensured.

   Our approach supports simulation of the generated control software on the development machine as well as execution on "real" ECUs. Using third-party software, simulation can be performed on a virtual model of the plant. This allows to synthesize and simulate control software before the concrete hardware is available, supporting hardware/software co-design [BCG+97, KL92].

   More information about the platform mapping can be found in Chapter 6.

1.4. Structure of this Thesis

This work is composed of eight chapters whose dependencies with regard to content are depicted in Figure 1.2. After this introduction, the thesis continues with background information about and current trends in the involved fields of research in Chapter 2,

namely industrial automation, embedded systems, model-driven development and game-based synthesis. In Chapter 3, a holistic overview of the workflow is depicted and the scope of this thesis is defined.

The work then continues with three chapters that are at the heart of this thesis: Chapter 4 shows how the considered class of automation systems as well as the tasks to execute on them is specified in a formal way. Chapter 5 presents how a solver that is based on game theory is used to derive concrete control programs from the abstract model. Chapter 6 shows how the synthesized control programs are mapped to one or more target platforms for simulation and execution. Relevant related work is listed in the respective chapters.

The last part of this work provides technical details about the implementation of the approach on a demonstration platform in Chapter 7. An evaluation of the system is provided as well. Finally, Chapter 8 summarizes the approach and the main results of this work and identifies directions for future research.

CHAPTER 2

Background and Trends

Contents

2.1. Industrial Automation **12**
2.2. Modeling and Model-driven Development **19**
2.3. Games and Game-based Synthesis **25**

Overview

This chapter provides an overview of three disciplines of research that serve as base technologies for the work presented in this thesis and draws conclusions on how this work can contribute to the state of the art. Those conclusions influence the design of the approach presented in the following chapters.

The first base technology is *industrial automation*. This chapter depicts current trends and provides an overview of state of the art hardware and software from the automation domain. Since this work employs a model-driven development approach, this chapter also covers the basic principles of *modeling and model-driven development*. It illustrates the difference between model-based and model-driven software development approaches and introduces typical modeling workflows. The last section of this chapter depicts relevant aspects of *game theory* and gives a formal definition of two-player games, which form the base of the synthesis approach that it used by this work.

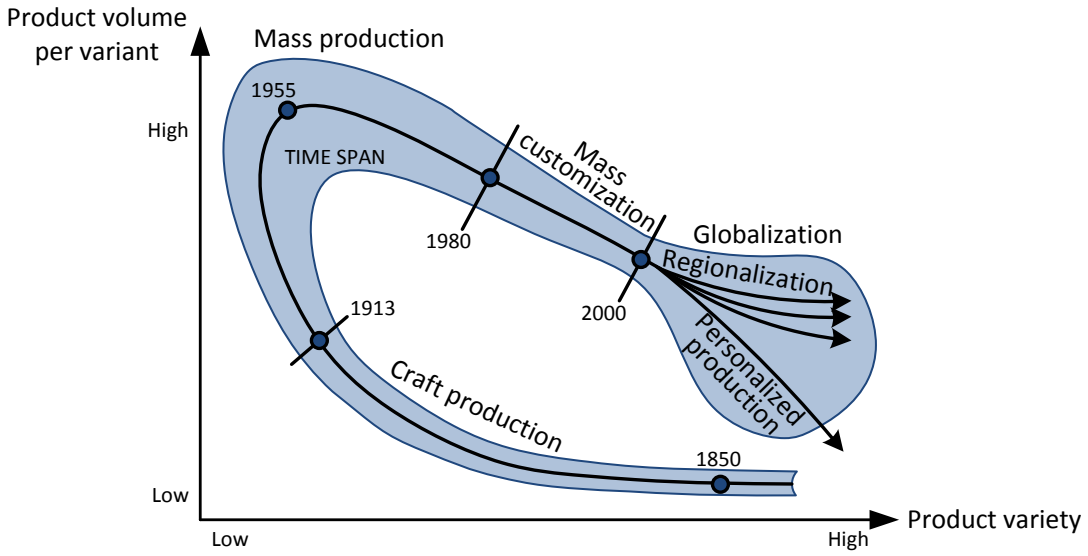


Figure 2.1.: Paradigm shifts in manufacturing over the past two centuries (adapted from [Kor10]): within this period, changing market demand and society needs triggered almost a "full circle" in manufacturing principles. The named years fit the paradigm shifts in automobile production in the Western world.

2.1. Industrial Automation

Industrial automation is about orchestration of a production plant in order to achieve a certain production goal. Over the past two centuries, it has become a vital part of manufacturing.

2.1.1. History of Manufacturing and Industrial Automation

The main drivers for the development of industrial automation are market demand and society needs [Kor10]. Due to changing market demand and society needs, the way in which industrial automation applies to manufacturing has been experiencing multiple paradigm shifts over the past centuries. This fact is illustrated by Figure 2.1 and summarized from [MUK00, Kor10] in the following.

Until the mid of the 20$^{\text{th}}$ century, **craft production** (i.e., the production of the exact product the customer asked for) prevailed and the use of automated equipment was limited to simple general purpose machines. Over time, market demand increased significantly. In order to achieve high production volumes, fully automated tasks were required to replace human workers. Since *dedicated manufacturing lines (DMLs)* were used for this purpose, which were only capable of producing a single product without variation, product variety decreased significantly, but the individual cost of goods was drastically reduced as well – the age of **mass production** had started.

Towards the end of the 20$^{\text{th}}$ century, the societal demand of customized goods grew permanently and product variety increased subsequently, while production volume

decreased slightly. In this age of **mass customization**, the automation systems needed to become flexible to maintain the low production cost of mass production. Hence, *flexible manufacturing systems (FMSs)* were designed to tolerate the variance introduced by customization (e.g., *computerized numerical control (CNC)* machines). An FMS is "a programmable machining-system configuration which incorporates software to handle changes in work orders, production schedules, part programs, and tooling for several families of parts. The objective of an FMS is to make the manufacture of several families of parts possible, with shortened changeover time, on the same system" [FFMV08].

Starting with the 21st century, **globalization** led to more competition as well as a higher volume of available products on the market due to **regionalization**. As a result, a trend towards differentiation based on **personalization** of products arose. Since the volume of personalized products being produced heavily depends on the customer-driven market demand at a specific point in time, FMSs, which typically have to cope with challenges such as high costs [KHJ+99], unreliability and obsolescence, are nowadays replaced by *reconfigurable manufacturing systems (RMSs)* [MUK00]. RMSs are designed in a modular way and can be reconfigured and upgraded quickly, both with respect to hardware and software. For example, a reconfigurable production line consisting of a linear arrangement of modular machines (so-called assembly stations) may be extended by adding the same types of machines in parallel to increase throughput if market demand is high. When market demand decreases later, the machines can be removed again and used elsewhere.

Koren summarizes the paradigm shifts mentioned above as follows [Kor10]:

> "Over the past two centuries, manufacturing has come nearly full circle: From focusing on the individual (Craft) to focusing on the product (Mass Production), to focusing on targeted market groups (Customization), and back to the individual customer (Personalization)."

Today, technological progress in industrial automation is characterized by research and development, but also by the adoption and continuous improvement of international standards and "de facto" standards such as [IEC03a, IEC07, Mod06]. These efforts aim at integration of the different layers of a production process with each other in order to form an efficient and manageable automated process. The following sections introduce the state of the art in industrial automation in greater detail and depict some of today's challenges.

2.1.2. Organization of Automation Systems

Industrial automation deals with the keywords *planning* (i.e., finding a strategy for realizing the production goal), *optimization* (i.e., ensuring that the plan is cost efficient), *data acquisition* (i.e., obtaining the state of the plant and making it accessible to *information technology (IT)* equipment, including error detection) and *control* (i.e., applying the plan to the physical control units in the plant under consideration of acquired data, including error handling).

Figure 2.2 shows a schematic diagram of the different layers in an automated plant, the so-called *automation pyramid* [IEC03c]. The shape of the pyramid illustrates the number

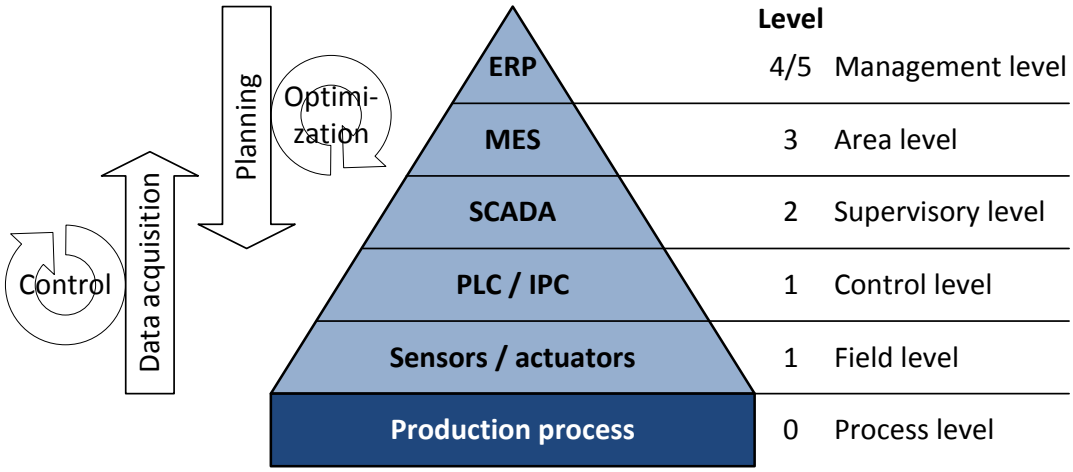


Figure 2.2.: Example of an automation pyramid with according level numbering [IEC03c]. Although IEC 62264-1 defines the control level or controller's level as level 1, the field level is commonly numbered the same way.

of entities present at the respective layers: while an automation system typically consists of at most one *enterprise resource planning (ERP)* system, it features a large number of sensors and actuators with hundreds or even thousands of input and output signals at the field layer.

Below the ERP system, there are typically one or more *manufacturing execution systems (MESs)*, which allow planning, real-time management, steering and control of overall production. MESs also take care of plan optimization, however suitability and flexibility is considered of higher priority than optimality [SM06]. *Supervisory control and data acquisition (SCADA)* systems provide monitoring and control facilities for specific processes. The actual control algorithms execute on *programmable logic controllers (PLCs)* (i.e., industrial grade special-purpose embedded systems) and *industrial personal computers (IPCs)*. In the following, this work uses the more abstract term *electronic control unit (ECU)* to refer to these components.

As indicated by the arrows on the left, planning and optimization is done at the higher levels of the pyramid and the results are propagated (e.g., in form of control programs or control commands) to the lower levels. Likewise, data that are acquired at the lower levels are continuously fed back to the higher levels in order to monitor the execution of the production plan and to detect and react to faults.

Propagation of information in either direction is carried out via dedicated communication technologies. Due to the number of signals and the proximity to the physical process to be controlled, the lower levels of the automation pyramid (levels 0 to 1) typically have (hard) real-time constraints with frequencies of 1,000 Hz or more (e.g., for synchronized motion control). Due to these requirements, special communication technologies are used, such as PROFIBUS (Process Field Bus) [IEC10], PROFINET (Process Field Network) [IEC10] or EtherCAT (Real-time Ethernet Control Automation Technology) [IEC07, IEC05b], to only name a few. The upper levels of the automation pyramid

(levels 2 to 4/5) typically work at a much lower update and control rate with fewer real-time requirements. Hence, cheaper solutions, such as *Internet Protocol (IP)* based Ethernet, are applied at those levels.

2.1.2.1. Programming

Most control software for automation systems in practice is designed for a unique or a specific type of system that exhibits recurring behavior. From a programmer's point of view, such tasks can be easily represented in a loop, where the body of the loop contains the primitive operations to execute. Hence, the goal is usually specified in an imperative manner as opposed to a declarative manner, where the functionality and extra-functional requirements of the system would be formally described.

This means that the resulting code or model is a mixture of the solution to achieve the given control task and the respective low-level control primitives. This makes such control programs rather inflexible, because the choice of commands and their order heavily depends on the requirements of the respective automation task. In case maintenance of the program is required afterwards, it is hard to reconstruct the original requirements. Robert Balzer once expressed this problem as follows: "[...] maintenance is performed on source code (i.e., the implementation). All of the programmer's skill and knowledge has already been applied in optimizing this form (the source code). These optimizations spread information; that is, they take advantage of what is known elsewhere and substitute complex but efficient realizations for (simple) abstractions" [Bal85]. Hence, manual adaptation of control programs bears the following risks:

1. Modification and subsequent integration and testing take a significant amount of time, increasing maintenance costs.
2. The specification for the existing system is not interpreted correctly, leading to unintentional changes to existing behavior.

Some of these problems have been addressed by using specific hardware and software tools. The next section introduces the most popular ones.

2.1.3. Hardware at Control and Field Level

The sensors and actuators in a traditional industrial automation system are controlled using ECUs. For this purpose, sensors and actuators are electrically wired to the ECUs on which the respective control software is running. In contrast to PLCs, IPCs typically also serve as front-ends for the physical process and run applications for data acquisition and logging, process visualization or monitoring (e.g., National Instruments LabView [TK06] or Siemens SIMATIC WinCC [Nan98]). They often feature a full consumer *operating system (OS)* such as Linux or Windows [SP03]. IPCs and PLCs can be expanded with "plug and play" capable expansion modules such as interface cards for specific fieldbus systems to customize them for the respective plant [Sie13]. In contrast to consumer *personal computers (PCs)*, IPCs offer higher guarantees with respect to reliability, compatibility, expansion and long-term support in order to cope with industrial requirements and environmental conditions.

A trend exists to decentralize data processing and process control in automation systems by introducing calculating capacity at the field level (compare Figure 2.2). This trend has started at least three decades ago when "minimum system cost [was] made possible by the adoption of suitable microcomputer systems for use as subsystems" [KKK84]. Among others, those subsystems are used to implement local control loops [OLK93]. In case of sensors, preprocessing is often directly integrated into the sensing component; the respective embedded systems are often referred to as *smart sensors*. Smart sensors are not as cheap as traditional sensors, but they significantly increase the level of service and maintainability, because they deliver preprocessed data and can be easily exchanged.

For example, in thermal production processes like tire production, preprocessing steps such as linearization, averaging and fusion of measured temperature or pressure values as well as measurement error correction are typically handled in smart sensors [IMB12, SGB+13, Fle12]. Furthermore, calibration can be externally triggered and correction coefficients are directly stored in the smart sensor's memory, allowing to exchange the sensor in-place without need for reconfiguration, provided that the replacement sensor has been calibrated in a laboratory before [Har12].

The application of smart, decentralized equipment as well as respective approaches for adaptability in a plant is commonly known as the trend towards the *smart factory* and also referred to as *factory of the future* [KLW11, aca11]. The German term "Industrie 4.0" (4^{th} industrial revolution [Sch12]) was coined to describe this trend. This term is very actively discussed in Germany [KLW11, Böh12a, Böh12b] and also subject of recent calls for research proposals [BMW12, Pro13].

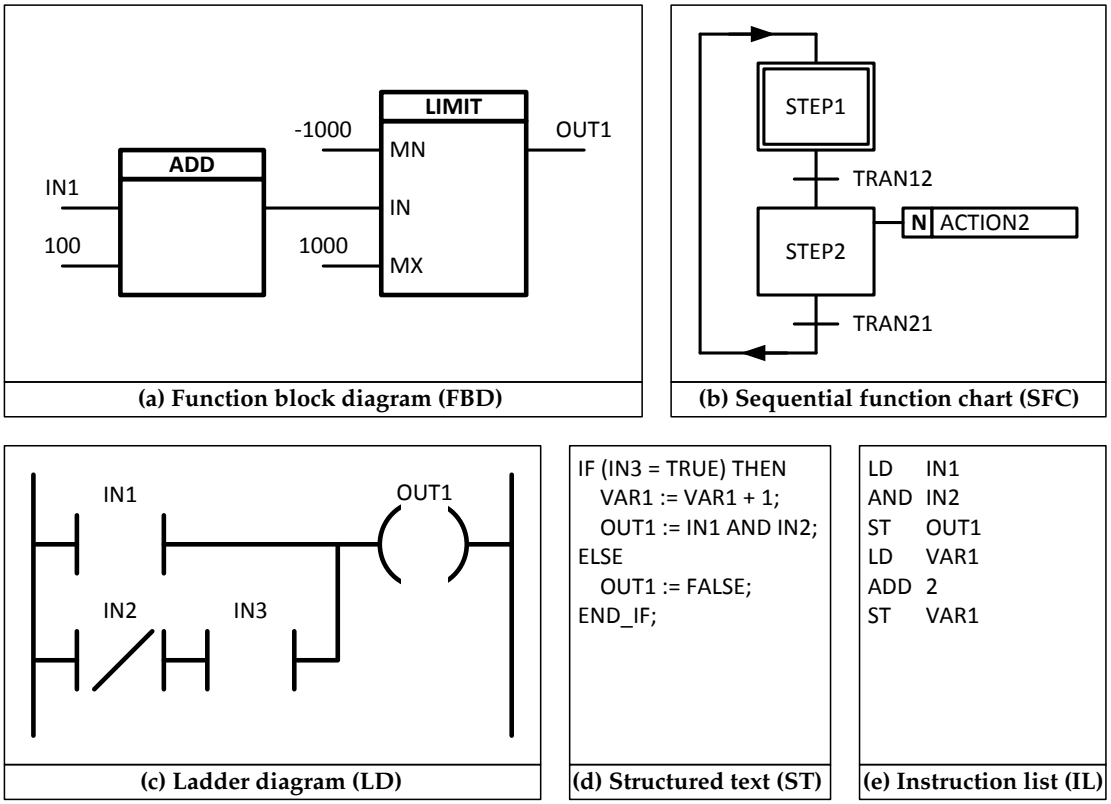
2.1.4. Software at the Control and Field Level

Programming of control algorithms running on IPCs (control level) is usually performed with traditional development toolchains (e.g., C/C++ compilers and *integrated development environments (IDEs)* [GS04, Abb08]), by code generation from automation specific software tools (e.g., from MATLAB/Simulink [CSV96] or National Instruments LabView [TK06]) or by using "soft PLCs" (i.e., the emulation of a PLC with support for PLC development tools).

Programming of (soft) PLCs (control level) is performed using domain-specific languages in dedicated programming environments. The international standard IEC 61131-3 [IEC03a] defines five[1] PLC programming languages. All of these languages share the concept of *program organization units (POUs)*. A POU can be a deterministic mathematical *function*, a *function block* or yet another PLC program. In this way, PLC programs can be nested. Figure 2.3 illustrates the syntax of each language with a simple example program.

- ***Function block diagram (FBD)*:** An FBD is a graphical language that describes a function between input variables and output variables (compare Figure 2.3 (a)). Formally, an FBD is a directed (multi-)graph in which vertices describe sub-

[1] Some sources do not count SFC as a PLC programming language, because it cannot resemble a functional program on its own (i.e., it rather models a state machine). This fact is neglected in this overview.



(a) Function block diagram (FBD) | **(b) Sequential function chart (SFC)**

(c) Ladder diagram (LD) | **(d) Structured text (ST)** | **(e) Instruction list (IL)**

Figure 2.3.: Common representation of (functionally unequal) programs in PLC programming languages [IEC03a]: (a) FBD representing the function $\mathrm{OUT1} = \min(\max(\mathrm{IN1} + 100, -1000), 1000)$; (b) SFC with two steps, two transitions and an action named ACTION2 associated to STEP2; (c) LD representing the function $\mathrm{OUT1} = \mathrm{IN1} \vee (\neg\mathrm{IN2} \wedge \mathrm{IN3})$; (d) ST fragment for the function $\mathrm{OUT1} = \mathrm{IN1} \wedge \mathrm{IN2} \wedge \mathrm{IN3}$ and a counter for the high-time of IN3; (e) IL fragment for the function $\mathrm{OUT1} = \mathrm{IN1} \wedge \mathrm{IN2}$ and an unconditional counter.

functions with respective input and output variables (called *ports*). Edges in the graph denote the *data flow* between output and input ports, between input variables and input ports or between output ports and output variables.

- ***Sequential function chart (SFC)*:** An SFC is a graphical language for modeling a specific class of finite state automata (compare Figure 2.3 (b)). Formally, an SFC is a directed graph with nodes corresponding to states and edges annotated with *transition conditions*. An SFC has a designated initial state which is active by default. A step becomes active when the steps above it are active and the respective transition condition is true, in which case the steps above it become inactive.
- ***Ladder diagram (LD)*:** An LD is a graphical language for rule-based modeling of *input/output (I/O)* relationships (relationships between *contacts* and *coils*). The name originates from the fact that a diagram in this language looks like a lad-

der with two vertical rails and rungs between them which represent the rules (compare Figure 2.3 (c)). During execution, matching input rules are applied in a quasi-parallel way. The respective output coils (i.e., variables, POUs or actuators) are set, executed or triggered, respectively.

- ***Structured text (ST)*:** ST is a text-based language with imperative execution semantics (compare Figure 2.3 (d)). The language is similar to Pascal [Bar81] and features control commands, mathematical and logical operations as well as invocation of POUs.
- ***Instruction list (IL)*:** IL is a text-based language with imperative execution semantics which is similar to assembly language (compare Figure 2.3 (e)). The control flow is determined by labels, jumps and function calls.

Two of the more popular IEC 61131-3 compliant software tools are:

- **Siemens SIMATIC STEP 7 [Jon06]:** Due to the market share of Siemens PLCs and IPCs in industrial automation (e.g., 19.5 % market share in the industrial sector in 2010 [IMS11]), the SIMATIC software family has evolved to one of the most common software tools for industrial automation. SIMATIC STEP 7 provides standard compliant design of industrial automation systems [Sie11] and also includes a large number of additional tools for development and simulation.
- **3S-Smart Software Solutions CODESYS [3S-14a]:** CODESYS (short for Controller Development System) is a hardware-independent and standard compliant framework for programming automation systems. It supports about 400 programmable devices from leading manufacturers from control engineering, automation components and embedded systems.

2.1.4.1. Decentralized Execution

Today, virtually all real-life automation systems are distributed systems that consist of a number of ECUs that cooperate in order to steer a production process. In order to meet the need to program distributed industrial control systems, the international standard IEC 61499 [IEC12] was created. The standard enhances the periodic execution model used in IEC 61131-3 by an event-driven concept. For this purpose, a generic function block model with event-driven inputs and outputs is introduced.

However, the distribution of control programs over a number of control units is typically a manual task: the control signals to exchange need to be manually specified, hence creating tight dependencies between *distributed control systems (DCSs)*. This means that when the distributed automation task needs to be adapted, the control programs on the affected ECUs need to be manually adapted.

For example, the task of processing a work piece in a set of processing modules is usually implemented by invoking control actions that move the work piece between the modules and by triggering the respective control actuations like drilling, grinding and testing (compare for example to approaches in vocational training [Fes12]). When the production process needs to be adapted by introducing a new processing step, at least the control programs for the neighboring modules need to be manually modified.

2.1.5. Conclusion and Consequences

The hardware technologies used in the industrial automation are very diverse. In order to cope with this "zoo" of technologies, international standards have been established that define common interfaces and development methodologies. Software engineering for industrial automation is based on special programming languages that use concepts and terminology from the control and electrical engineering domains in order to allow programming of automation systems by the developers from these domains.

However, the level at which the programming occurs is a rather low one: programming occurs at function level, while interconnection occurs at the level of logical I/O signals. This is desirable to be able to control each and every aspect of the system. But, especially in the case of distributed systems, the orchestration of the individual ECUs becomes a tedious task.

To cope with these issues, **a tool-supported workflow for the development of distributed control programs that raises the level of abstraction beyond the current level of implementation** is proposed in this work. This workflow integrates the existing hardware and software tools. In this work, the approach is restricted to a specific class of automation systems where the potential gain is significant in terms of flexibility and maintainability. This class of automation systems is further referred to as *modular assembly lines (MALs)*. For details, see Section 3.3.1.

This section showed that control engineers use special languages and tools in order to develop the application logic for automation systems. Since our goal is to use an even higher level of abstraction for both description of automation systems as well as task specification, we need adequate tool support as well. In addition, for programming embedded systems, we need mechanisms to automatically generate code and system configurations from the tools. This approach, which is commonly referred to as *model-driven development*, is introduced in the next section.

2.2. Modeling and Model-driven Development

2.2.1. Modeling

The purpose of *modeling* in general is to construct an artifact "that resembles in some way an imagined or existing system or process. Models serve as surrogates of the system or process that they represent in order to help us understand or appreciate it more" [Sel11b] or to derive a concrete system implementation from it. "The intent of modeling is to reduce the full scale of the represented phenomenon to something accessible to human comprehension or some type of formal treatment" [Sel11b]. For this purpose, the model, unlike the imagined or existing system, shall contain only the information necessary to achieve a designated task. Models do not only "help us understand the represented phenomenon" [Sel11b], but also help to "communicate our understanding to others" [Sel11b].

Although the general concept has been around for a long time, modeling is still a rather

young scientific discipline. Bran Selić, who has been involved in the definition of the *Unified Modeling Language 2 (UML2)* standard, stated in 2009 [Sel11b]:

> "The design of modeling languages is still much more of an art than a science. There is as yet no systematic consolidated body of knowledge that a practitioner can refer to when designing a computer-based modeling language."

In the context of computer software, modeling is a *software engineering* approach. When we introduced programming tools and languages for industrial automation in Section 2.1.4, we actually introduced a domain-specific modeling approach. Similarly, when a programmer implements an application in his favorite high-level programming language, the program is a model of what he or she intends to achieve that needs to be mapped to the target system for execution by compilation or interpretation. This is most obvious for programming languages whose execution is based on virtual machines (such as Java [MCF03]), but also true for many others.

Modeling is essential for adequate programming of automation systems, because it would otherwise be virtually impossible for a control engineer with little or no background in computer science to program the system. Modeling introduces a suitable degree of abstraction, hides unnecessary implementation details and may provide automatic mechanisms to make the models executable. This reduces the overall complexity and allows the developer to focus in the actual task to achieve.

2.2.2. Model-based vs. Model-driven

Processes around modeling can be roughly separated into *model-based design/development (MBD)* and *model-driven development (MDD)*; the latter is also called *model-driven engineering (MDE)*. Both involve the application of modeling concepts for software development by…

1. creating *domain specific languages* to represent selected properties of a particular application domain,
2. using these languages as metamodels for specifying concrete models (so-called *domain specific modeling* [KT08]) and
3. using the models as system specification, for validation, verification and/or documentation.

In contrast to *model-based* approaches, *model-driven* approaches…

4. synthesize source code from the respective models, too.

Figure 2.4 illustrates how model-based and model-driven engineering compare to traditional software development approaches. Assembly language can be seen as the approach with the least degree of computing technology abstraction. Classical so-called third-generation programming languages (e.g., C/C++ [KR88, Str13], Java [AGH06], Pascal [Bar81]) have a higher degree of abstraction and use compiler toolchains to bridge the gap to the implementation level.

In contrast, modeling languages typically use a much higher level of abstraction. For further processing, models are typically validated and/or transformed. Model *valida-*

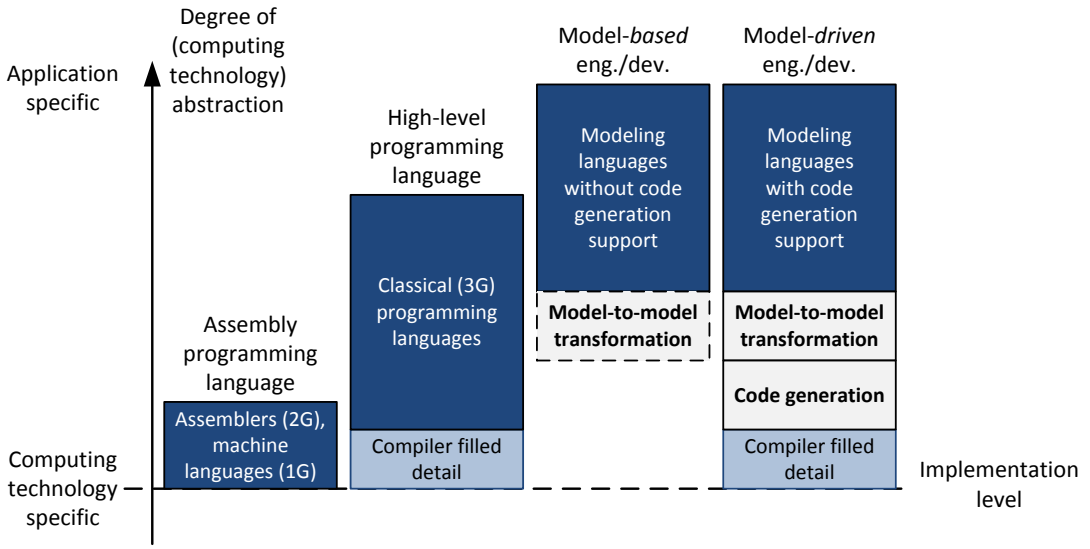


Figure 2.4.: Computer programming and modeling language abstraction (adapted from an illustration by Bran Selić [Sel11a]): the so-called 1G and 2G languages have limited expressiveness, but allow programming a system directly at the implementation level. 3G languages use compiler toolchains to fill the gap to the implementation level. Modeling languages use a high level of abstraction, but only model-*driven* approaches potentially allow a transformation to executable code at the implementation level.

tion is performed to check the consistency of the model. Model *transformation* combines information from multiple models (e.g., to form a composed model) or generates new information from a model (e.g., calculation of a schedule from task specifications).

Model-*based* approaches are typically used for documentation purposes or to better understand the system under inspection. They are suitable to visualize design concepts and are very common in software engineering. In model-*driven* approaches, there exists a process that maps the model to the implementation level. For this purpose, the models contain properties for mapping their abstract entities to the concrete target system.

2.2.3. Domain-Specific Languages

A *domain-specific language (DSL)* is a programming language dedicated to a specific problem domain. It includes all concepts (and usually no more) that are necessary to formally specify all relevant aspects of a problem in that domain, including the constraints under which the problem is to be solved. A DSL can be seen as the opposite of a general purpose modeling language such as the *Unified Modeling Language (UML)*, which is domain independent. The systematic use of DSLs to specify problems is called *domain-specific modeling (DSM)*. A DSM can be used to automatically generate source

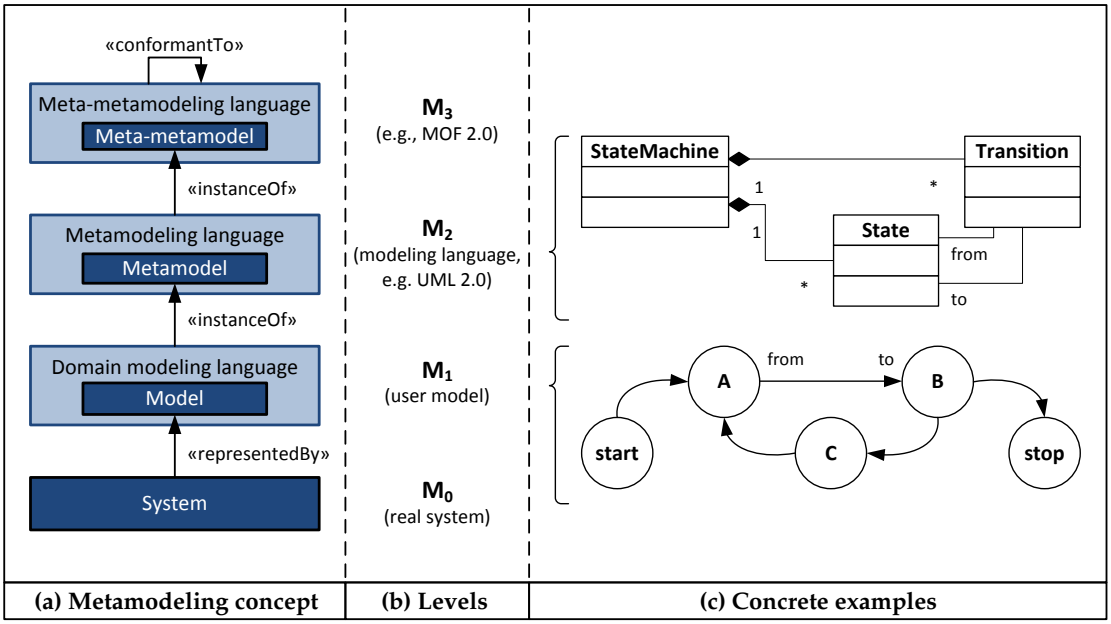


Figure 2.5.: (a) Metamodeling concept with according level names (b): at levels M_3 and M_2, a "meta"-model is provided that defines the building blocks for the level below it. The model at level M_3 conforms to building blocks from M_3 in order to terminate the recursive definition. (c) Concrete examples (adapted from [OMG11b, ISI05, Buc08]): the behavior of a real system is represented by a finite state machine, which consists of the entities StateMachine, State and Transition. These concepts are in turn based on MOF.

code corresponding to the concepts in the model. "Early adaptors of DSM have been enjoying productivity increases of 500 – 1000 % in production" [KT08].

Despite the fact that the term "programming language" is usually understood as a text-based format, a DSL is not necessarily text-based. In contrast, the strength of many DSLs consists in the fact that they are intuitive to end users due to their graphical structure or denotational semantics (e.g., MATLAB/Simulink [The13]).

The components of a DSL that serve as the building blocks for the respective model instance are commonly referred to as the *metamodel*. Those need to be in turn formally defined by a language of their own. Since the DSL is the metamodel, the language in which it is written in is often referred to as a *meta-metamodel*.

Figure 2.5 illustrates the concept of metamodeling: at level M_3, a generic meta-metamodel is provided that defines basic constructs for defining metamodel elements one level below, such as containers and relations. For example, *Object Management Group's (OMG) Meta-Object Facility (MOF)* [OMG11a, IEC05a] defines a suitable base language. Level M_2 uses these constructs to define a DSL that serves as a metamodel for the actual model at level M_1 in the hierarchy. That model in turn represents the real system at level M_0.

2.2.4. Model Validation

Most models can be in an inconsistent state. The reason for this is that in most cases, not all desired constraints can be formulated by just the syntax of the modeling language itself. For example, consider the finite state machine metamodel from Figure 2.5 (c): this metamodel allows defining a state machine in which not all states are connected by transitions. Since it may not desirable to allow such a state machine to be defined, it is meaningful to *validate* the model before it is further processed. This allows detecting design errors in an early phase of development.

Validation takes place by checking whether all *check constraints* specified for the model are satisfied. Due to the structured nature of the models, it is usually easy to specify such invariants. Check constraints are typically formulated in special languages such as the *Object Constraint Language (OCL)* [OMG12a] or Check, where tests against the model are specified in first order logic [Buc08]. The purpose of these languages is to verify properties of the models and to generate diagnostic messages in case of constraint violations while typically not allowing any modifications of the models itself. Hence, such languages are typically of functional nature without any side effects. In the above example, a meaningful informal check constraint would be $\forall s_1, s_2 \in$ set of States of the same StateMachine and $s_1 \neq s_2$, there must exist a path from s_1 to s_2 using Transitions from that StateMachine (ignoring the "direction" of the transitions).

2.2.5. Model-to-model Transformation

Especially in model-driven engineering, where the goal is to map a model to the implementation level, a step-wise transformation of the model created by the end user is meaningful according to "divide and conquer" principles. For example, if a model specifies software components that need to be executed on a set of distributed target platforms, a process that would directly derive the code to implement the respective programs from the top-level model would be monolithic, complex and hard to maintain.

Instead, so-called *model-to-model transformations (M2Ms)* are added that adapt the model step by step until it is ready to be transformed into executable code. In the above example, M2Ms would first generate the required schedules for the execution of the software components, then distribute the software components over the set of target platforms and finally add platform-specific implementation details.

The goal of applying M2Ms is to generate all information required for code generation such that code generation itself can be formulated as a direct transformation from model to text. This allows a clear separation of concerns between algorithmic parts and code generation parts and fosters reusability and maintainability.

2.2.6. Model-to-text Transformation and Program Synthesis

Since a model at level M_1 is an instance of elements from the metamodel at level M_2, which in turn are based on primitives with well-known semantics on the meta-metamodel at level M_3, an automatic process can be used to generate executable program code (in an arbitrary 3G, 2G or 1G language, compare Figure 2.4) from the model. In addition, code for representing, storing (serializing/de-serializing), viewing and editing a model (e.g., according to *model-view-controller* pattern [Bur87]) can be generated from its metamodel. Such processes are commonly known as *automatic programming* [Bal85].

Using this approach, program parts that are time-consuming to build manually are generated automatically with the added value that, provided model and code generation are algorithmically correct, the resulting code is *correct by construction*. The code is often generated from a respective *template* that is itself written in a DSL in which some parts are simply copied to the resulting generated file and specially marked parts are replaced by elements from the respective model (e.g., template expressions in Xtend [Ecl14]). This approach allows designing the code generation algorithm in a more generic way, because it does not need to know the exact semantics of the code being produced. Instead, the code generator interprets the template language in order to produce the final result.

Notice that the generated "code" does not have to be program code. For example, a typical use case is to generate reports and documentation from models, hence targeting human readers. For this purpose, the more general term *model-to-text transformation (M2T)* is typically used instead of the term "code generation".

Program synthesis is a more formalized kind of automatic programming. "Program synthesis is concerned with the following question: given a not necessarily executable *specification*, how can an executable program satisfying the specification be developed? The objective of program synthesis is to develop methods and tools to mechanize or automate (part of) this process" [BDF+04].

Hence, in contrast to code generation, synthesis does not only deal with a direct transformation of a model to code. Instead, the model poses a formal "problem" in the context of the DSL defined by the respective metamodel. The goal is to find a feasible "solution" for the formal problem that honors the domain-specific constraints set up in the model. If a solution exists, the result is another model – typically in a different DSL – that specifies a solution to the problem. Synthesis might fail If no solution exists or could be found (e.g., because the problem is undecidable).

2.2.7. Existing Modeling Tools and Frameworks

Modeling tools can be roughly separated into general purpose modeling tools, domain-specific language frameworks and domain-specific modeling environments.

The most popular general purpose language is the *Unified Modeling Language (UML)* [OMG11b, OMG11c]. There are many tools available to create UML-compliant models and to generate code from them. Examples are IBM Rational Rhapsody (com-

mercial closed source) [IBM12], the Eclipse *Unified Modeling Language 2 (UML2)* Tools (open source) [Ecl13] and Sparx Systems Enterprise Architect (commercial closed source) [Spa13], to only name a few. Many of these tools also support UML profiles such as SysML [OMG12b] or MARTE [OMG11d].

Popular *domain-specific language frameworks* (for designing DSLs) include the *Eclipse Modeling Project (EMP)* with *Eclipse Modeling Framework (EMF)* and *Graphical Modeling Framework (GMF)* (all open source) [BSM+04, Gro09], MetaEdit+ (commercial closed source) [Met13] and actifsource (commercial closed source with free community edition) [act12].

Well-known examples for domain-specific modeling environments (besides those listed in the context of industrial automation in Section 2.1.4) are commercial tools such as MATLAB/Simulink [CSV96] for implementation of control theoretic algorithms or Esterel Technologies SCADE [SGT+12] for development of safety critical applications. There are also many scientific domain specific modeling environments available, for example Ptolemy/Ptolemy II [EJL+03, BL10] for modeling, simulation, and design of concurrent embedded real-time systems, Giotto for programming "embedded control systems running on possibly distributed platforms" [HHK01] and EasyLab for graphical design of control programs for microcontrollers in mechatronic applications [BGBK08, BGH+10].

2.2.8. Conclusion and Consequences

Modeling is *the* key tool for domain experts such as control engineers to develop complex systems. This is why this work introduces a **model-driven development tool** that allows describing the structure and topology of an automation system as well as the task to achieve with it in a **domain specific language**. Based on this information, model-to-model and model-to-text transformations are carried out in order to obtain a suitable control program. The Eclipse Modeling Framework has been used for implementing the tool that has been developed along with this work. Reasons for this choice are that it is freely available, open source and well maintained.

An important step in the workflow described in this thesis is the generation of a suitable control algorithm for a given model. The last section of this chapter introduces game-based synthesis, which is the formal base of this approach.

2.3. Games and Game-based Synthesis

The *game theory* is a research area from theoretical computer science. It uses mathematical methods to make strategic decisions in a given problem space in order to reach a certain goal. This overview is restricted to *turn-based games*, i.e., games where the players make moves by taking turns. There exist many types of turn-based games for different numbers of players: one player games (e.g., sliding puzzles) ask to find a strategy to solve a given problem, where each move changes the state of the game. A *strategy* is a description of the decisions a player will make at all possible situations that can arise in the game [Tho84]. *Two player games* (e.g., chess) involve two opponents

with different (typically opposite) goals. The player that manages to reach his/her goal and/or prevents the other player from reaching his/her goal wins the game.

Besides the number of players, games can have many different properties. In games with perfect information (e.g., chess), the players can observe the whole state of the so-called *arena*. The challenge of such games lies in the computational complexity of exploring all possible moves. In contrast, games with imperfect information (e.g., most types of card games) hide some information from the players [Tho84] and hence demand decisions based on probability.

In the following, *two-player games* are formally introduced, which form the base of the synthesis approach presented in this work.

2.3.1. Formal Definition of Two-player Games

The following definitions have been adapted from [Che12] and [GTW03]. Formally, a *game* $\mathcal{G}$ is a pair

$$\mathcal{G} = (\mathcal{A}, \mathcal{W}) \tag{2.1}$$

where $\mathcal{A}$ is an arena and $\mathcal{W}$ is the winning set. In *a two-player game*, an *arena* is a triple

$$\mathcal{A} = (\mathcal{V}_0, \mathcal{V}_1, E) \tag{2.2}$$

where $\mathcal{V}_0$ is a set of *0-vertices* and $\mathcal{V}_1$ is a set of *1-vertices* with $\mathcal{V} = \mathcal{V}_0 \uplus \mathcal{V}_1$ (i.e., $\mathcal{V}_0 \cap \mathcal{V}_1 = \emptyset$). The arena is assumed to be finite (i.e., $|\mathcal{V}| < \infty$). $E \subseteq \mathcal{V} \times \mathcal{V}$ is the *edge relation* also called the *set of moves*. Hence, $\mathcal{V}$ and E define a directed graph. The set of *successors* of $v \in \mathcal{V}$ is defined by

$$vE = \{v' \in \mathcal{V} \mid (v, v') \in E\} \tag{2.3}$$

In the two-player game, $\mathcal{V}_0$ are the vertices in which *player 0* takes turns and $\mathcal{V}_1$ are the vertices in which *player 1* takes turns. A *token* is initially placed on one of the vertices $v_0 \in \mathcal{V}$. If v_0 is a 0-vertex, player 0 makes the first move; otherwise player 1 makes the first move. During a player's turn, the player moves the token from the vertex v it is currently located at to a successor $v' \in vE$ of v. If v' is a 0-vertex, then player 0 continues, otherwise player 1, and so on. A *play* π in the arena $\mathcal{A}$ is a finite or infinite path according to the above rules. If the path is finite, formally

$$\pi = v_0 v_1 \dots v_l \in \mathcal{V}^+ \text{ with } v_{i+1} \in v_i E \text{ for all } i < l, \text{ but } v_l E = \emptyset, \tag{2.4}$$

then the play is called a *finite play*. If the path is infinite, formally

$$\pi = v_0 v_1 v_2 \cdots \in \mathcal{V}^\omega \text{ with } v_{i+1} \in v_i E \text{ for all } i \in \omega, \tag{2.5}$$

where ω is the set of non-negative integers (i.e., $\omega = \{0, 1, 2, 3, \dots\} = \mathbb{N}_0$), then the play is called an *infinite play*. $\mathcal{W} \subseteq \mathcal{V}^\omega$ is the *winning set* of a game. Player 0 is declared the *winner* of a play π if and only if

- π is a finite play $v_0 v_1 \dots v_l \in \mathcal{V}^+$ with $v_l E = \emptyset$ and $v_l \in \mathcal{V}_1$ (i.e., no moves are left for player 1, because he has reached a *dead end*) or
- π is an infinite play and $\pi \in \mathcal{W}$.

Player 1 wins π if player 0 does not win π.

2.3.2. Games and Industrial Automation

Control problems from industrial automation can be encoded in two-player games, where a winning strategy for the game corresponds to the successful execution of an automation task. For illustration, think of an automation task as a chess game [GC13]: the moves of the player with the white pieces correspond to the production steps carried out in the automation line and the moves of the player with the black pieces correspond to the respective reactions of the environment. In one of its moves, White (the control program or *Controller*) could for example "ask" for a property of a work piece in the automation line (i.e., the triggering of a sensor) and Black (the *environment*) would "answer" this question on form of one of its legal moves. The set of legal moves changes depending on the state of the game.

Controller's moves include the triggering of actuators as well as the triggering of sensors. Environment's moves allow returning a certain value from a sensor triggering. By extending the semantics of a sensor triggering, this approach can also be used to model the injection of faults into the system. In order to guarantee to win the game, White must be able to cope with any of the moves that Black takes.

Just like two chess players need to plan their moves strategically and adapt to the moves of their opponent, the automation system needs to react to the properties of work pieces, the advancing of the production process and potential faults in order to successfully complete the production task. In the figurative sense, the goal is to guarantee to checkmate Black, i.e., to win the game no matter which moves Black makes. White winning the game means successful completion of the automation task.

Notice that the analogy with chess is just used for illustration. The suggested approach is only meaningful if the complexity of the game is much lower than the one of chess, because the time it takes to find optimal chess moves is in general unacceptably long. However, provided certain assumptions are made, the complexity of the "game" between *Controller* and *environment* is low enough to be able to synthesize a suitable control program within seconds to minutes (compare evaluation results in Chapter 7).

2.3.3. Game-based Synthesis

The essence of *synthesis* is described as follows: "Consider a system consisting of a process, an environment and possible ways of interaction between them. The synthesis problem is stated as follows: given a specification S, find a finite state program P for the process such that the overall behavior of the system satisfies S, no matter how the environment behaves." [MW03]. Provided that the specification is correct, synthesis is *correct by construction*.

This definition of synthesis fits well the game-based scenario with two players: the process (i.e., Controller) corresponds to *player 0* and the environment corresponds to *player 1*. Interactions between the process and the environments are represented by moves (i.e., interactions) of the respective players in the game. Synthesizing a control program for player Controller equals to finding a finite state program/machine P that wins the game satisfying specification S no matter how player Environment behaves.

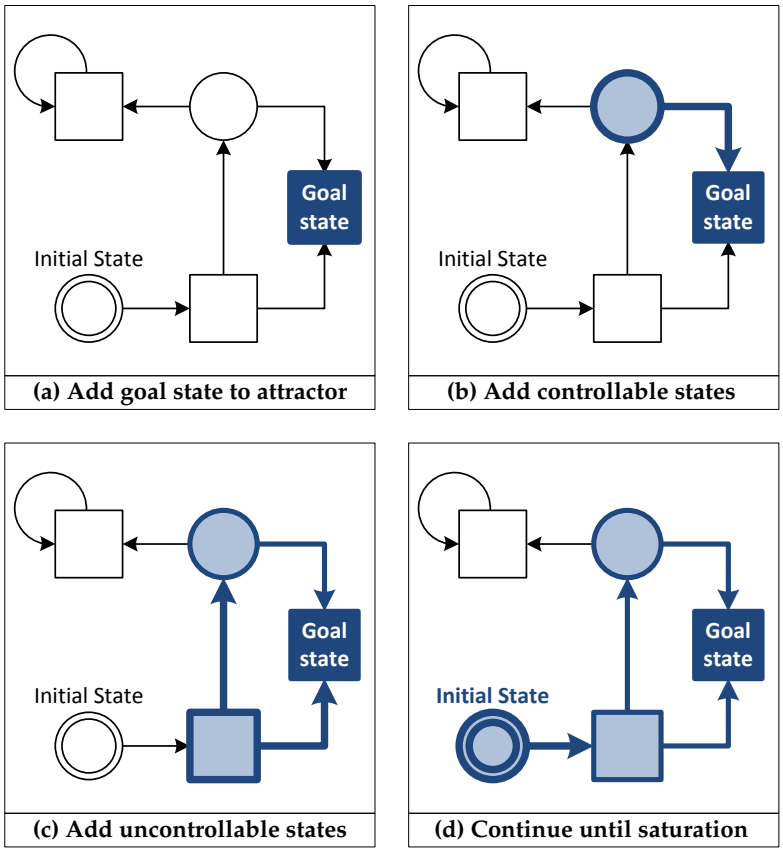


Figure 2.6.: Example for game-based synthesis with two *0-vertices* (cicles) and three *1-vertices* (squares): the so-called *attractor computation* calculates the states in which Controller can force to win the game.

Figure 2.6 illustrates how synthesis works for two-player games on finite graphs: the set of "circle" vertices corresponds to $\mathcal{V}_0$ and the set of "square" vertices corresponds to $\mathcal{V}_1$. The so-called *attractor computation* calculates the set of vertices in the game graph that, once visited, guarantee Controller to reach the goal state. This set is calculated as follows:

1. Let $\mathcal{V}_a \subseteq \mathcal{V}$ with $\mathcal{V}_a := \emptyset$ denote the initially empty set of vertices in the attractor.
2. Add all goal states to $\mathcal{V}_a$, because if a goal state is visited, Controller wins the game (compare Figure 2.6 (a)).
3. Add a *0-vertex* $v_0 \in \mathcal{V}_0$ to $\mathcal{V}_a$ if it has an edge that leads to a vertex $v' \in \mathcal{V}_a$, because when in v_0, Controller can select that edge (compare Figure 2.6 (b)).
4. Add a *1-vertex* $v_1 \in \mathcal{V}_1$ to $\mathcal{V}_a$ if all its edges lead to vertices in $\mathcal{V}_a$, because in this case Environment must choose a move that leads to $\mathcal{V}_a$ (compare Figure 2.6 (c)).
5. Repeat the above two steps until saturation (compare Figure 2.6 (d)).
6. If the initial state is in $\mathcal{V}_a$, then Controller is guaranteed to win the game. Otherwise, Controller is not guaranteed to win.

2.3.4. Distributed Synthesis

In a distributed system with finite state, multiple ECUs exist. The problem of *distributed synthesis* is to find control programs $P_1, P_2, \ldots, P_k$ for each of the ECUs such that the overall behavior of the system satisfies S [MW03]. Such control programs are referred to as *decentralized control programs*. It has been shown that distributed synthesis is undecidable in general [PR90] even for reachability or simple safety conditions [Jan07]. However, distributed synthesis is elementarily decidable for some distributed architectures (architecture here refers to a set of processors with corresponding shared variables and allowed communication patterns). Furthermore, the decomposition of a global control program into individual control programs is decidable for acyclic architectures [PR90].

This work applies a decomposition approach on a synthesized global control program in order to obtain individual control programs for every ECU. This methodology is not complete, because it may not find a realization even if S is realizable over the respective architecture [PR90]. Decomposing a global control program is suitable and meaningful in the target domain of this work, because the decomposed programs can be as easily understood by humans as the respective global control program. In addition, this approach allows switching between central and distributed control by applying or omitting the decomposition step. Last but not least, synthesizing an *optimized* control program is often considered enough for the application domain of industrial automation as opposed to synthesizing *optimal* distributed control programs [SM06].

2.3.5. Conclusion and Consequences

Modeling an automation task as a two-player game is a suitable way to enable mathematically sound treatment. This approach allows to synthesize a suitable control program for a given task specification. Furthermore, distributed synthesis can be applied to obtain individual control programs for a set of ECUs that need to cooperate in order to achieve a global automation task.

The next chapter combines model-driven development and game-based synthesis into a workflow for control software development in industrial automation.

CHAPTER 3

Overview of the Approach

Contents

3.1. User Roles and Domain Knowledge 32
3.2. Derived Workflow 33
3.3. Scope of this Work 37
3.4. Introduction of the Running Example 39

Overview

This chapter provides a holistic view of the workflow for programming *modular assembly lines (MALs)* that is presented in this work. The workflow is designed according to the conclusions from the previous chapter.

First, the types of users that are involved in the design of control software for industrial automation systems are listed. They are subdivided into roles based on the domain knowledge of the respective users. Subsequently, a confluent workflow is defined in which even users with no expert knowledge in industrial automation can specify and adapt the high-level specification of the plant and the task to execute. This concept is not feasible in traditional development approaches, where expert knowledge is required in both setup and adaptation of automation system software.

Finally, this chapter defines what is in the scope of this thesis and what is not and introduces a running example that is used to illustrate the developed workflow in the following chapters.

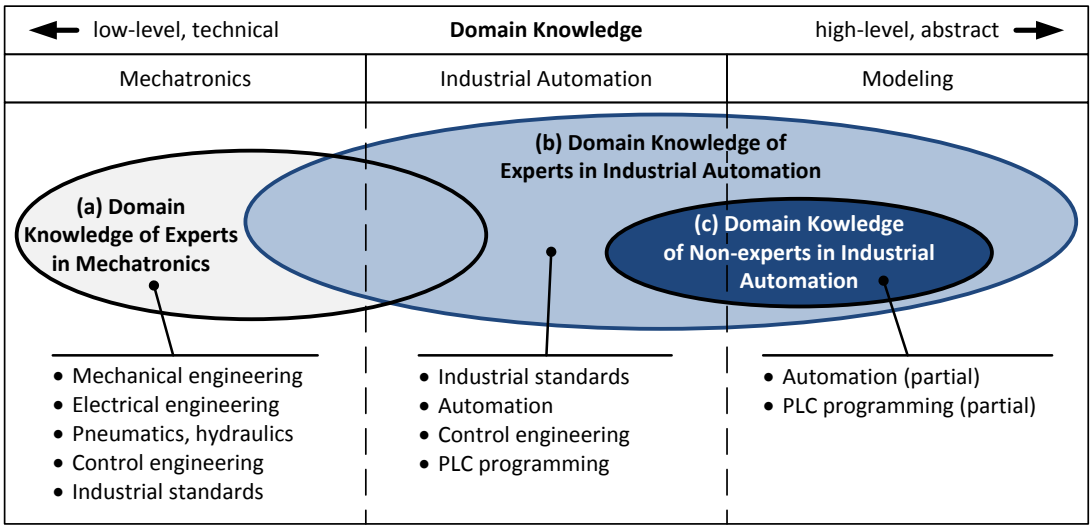


Figure 3.1.: User roles and domain knowledge in control software design: we distinguish three roles of users with different domain knowledge that are involved in industrial control program design: (a) *experts in mechatronics* have profound knowledge about mechanical and electrical engineering, while (b) *experts in automation* are more concerned with assembling automation systems and designing the respective control algorithms. (c) The domain knowledge of *non-experts in automation* is a subset of the knowledge of the latter group.

3.1. User Roles and Domain Knowledge

Throughout this work, it is assumed that different groups of users with different technical backgrounds contribute to the software development process for industrial assembly lines. The technical background of a user or group in its area of profession is further referred to as the *domain knowledge*. The individual roles are characterized in the following.

Experts in Mechatronics

An *expert in mechatronics* (Figure 3.1 (a)) is a person or company with a strong background in electronics and mechanics. Experts in mechatronics supply the hardware modules that are used in an industrial assembly line. They understand the requirements that users of such hardware modules have. Their work is characterized by the analysis of the automation tasks to solve, selection of the appropriate implementation (e.g., mechanic, electric, pneumatic or hydraulic mechanisms), assembly of the according hardware modules and offering of the modules as products. These modules are then provided or sold to experts in industrial automation.

Typical members of this group are employees in companies who build automation components that are to be used in a medium or large scale automation system (e.g.,

Festo AG) or special departments in a manufacturing company that are concerned with building automation systems for use in the company's own production plants.

Experts in Industrial Automation

An *expert in (industrial) automation* (Figure 3.1 (b)) is a person or company with a strong background in automation and control engineering who analyzes which tasks are to be automated in a production scenario or receives respective requirements from a stakeholder. He or she then plans and conducts the realization of an automation system that fulfills these requirements. The automation system is built from hardware modules provided by experts in mechatronics. In addition, an expert in automation selects suitable *electronic control units (ECUs)* to connect to the hardware modules in order to control the automated process. Furthermore, he or she designs and implements the software that needs to run on the ECUs in order to fulfill the specification.

Typical members of this group are employees with many years of experience in their profession that are either involved in building automation systems for other companies (e.g., Bosch Rexroth AG, Rockwell Automation, Inc.) or are responsible for the automation systems in their own manufacturing company, because the production processes have a high degree of automation (e.g., large car manufacturing companies).

Non-experts in Industrial Automation

In this work, an additional user role is defined that we call *non-expert in (industrial) automation* (Figure 3.1 (c)). Such a role corresponds to a person or company with a less profound background in automation and control theory than an expert in (industrial) automation. He or she has some knowledge about the structure of automation systems and the tasks that can be achieved by them, but he or she has no in-depth knowledge on how to build an automation system or how control software for such systems is implemented, such as IEC 61131-3 programming languages.

However, a non-expert in automation is able to recognize properties of the work pieces to be processed in the automation line and understands how requirements for production are explicitly specified in a high-level language.

Typical members of this group are students and trainees in industrial automation as well as users from other disciplines. Furthermore, *small and medium enterprises (SMEs)* that want to automate parts of their production, but cannot afford to employ an expert in automation, are associated with this category.

3.2. Derived Workflow

3.2.1. Traditional Bottom-up Workflow

Figure 3.2 outlines the traditional development process for industrial automation system software. It is a refinement of Figure 1.1 (a) on page 7. The lower half of the figure shows that domain-specific tooling exists that allows developers to program the system

in languages such as *programmable logic controller (PLC)* programming languages (compare Section 2.1.4). *Primitive control functions (PCF)* are implemented in these languages in form of function blocks. The combination of function blocks forms the actual control program. Hence, this development process is a *bottom-up* approach.

Although implementation details of the system are more and more abstracted when moving vertically up the diagram, the individual steps require knowledge about the behavior of the lower levels, because properties such as the actual topology of the plant and real-time requirements are often implicitly encoded in the control program. This means that expert knowledge is required to design and adapt the control software.

3.2.2. Proposed Confluent Workflow

In order automate parts of the development process of industrial control systems, we decompose it into discrete steps with defined specification of the interfaces between those steps. This enables us to exchange some of these steps by automated processes.

In the introduction to this thesis, we mentioned already that the development process for control software is split into three steps in order to reach the intended goals (compare Figure 1.1 (b) on page 7). The reason for this split is guided by **Goal 1** and **Goal 2** as presented in Section 1.2: three individual steps are the minimum number of steps to separate the development process into

- a manual part that requires experts in mechatronics and industrial automation,
- a manual part that does not necessarily require experts in industrial automation
- and a part that consists of fully automated transformations.

This section refines these steps and concretizes the workflow behind them. Figure 3.3, which is a refinement of Figure 1.1 (b), illustrates these steps. An important characteristic of the proposed workflow is its consistency and continuity: it includes all aspects from specification of the plant model to simulation and execution of the control programs on concrete target hardware.

3.2.2.1. Generic Control Primitives

Just as in the traditional development process, a domain expert uses existing tools (such as IEC 61131-3 PLC programming languages) to program the low-level behavior of the individual components in an automation system. However, these programs are not directly associated to a specific automation system. Instead, they are designed in a generic way and disclose interfaces that are used to parameterize and trigger the respective actions. For example, a conveyor belt offers an interface to transport work pieces and a drill offers a (parameterized) interface for drilling a work piece.

These so-called *behavioral primitives* are similar to drivers in an *operating system (OS)*: they make a certain hardware module accessible to user-level programs and provide a generic interface that does not unnecessarily limit the potential applications. A similar approach is applied by CODESYS Application Composer [3S-14b], which is used to build application software for automation systems from recurring function blocks.

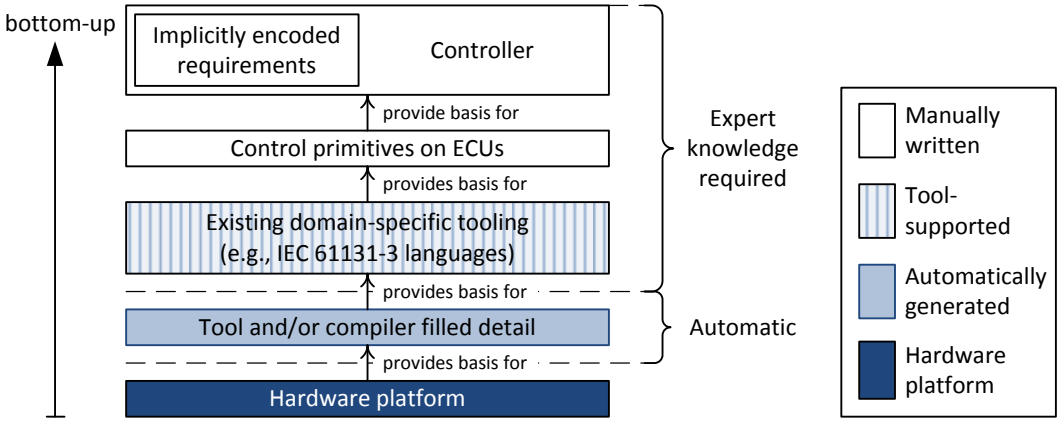


Figure 3.2.: Traditional *bottom-up* control program design: domain-specific tooling is used to build and combine basic function blocks in order to implement a control program that specifies *how* the respective task is achieved. Requirements such as the actual topology of the plant and extra-functional properties are implicitly encoded in the control program.

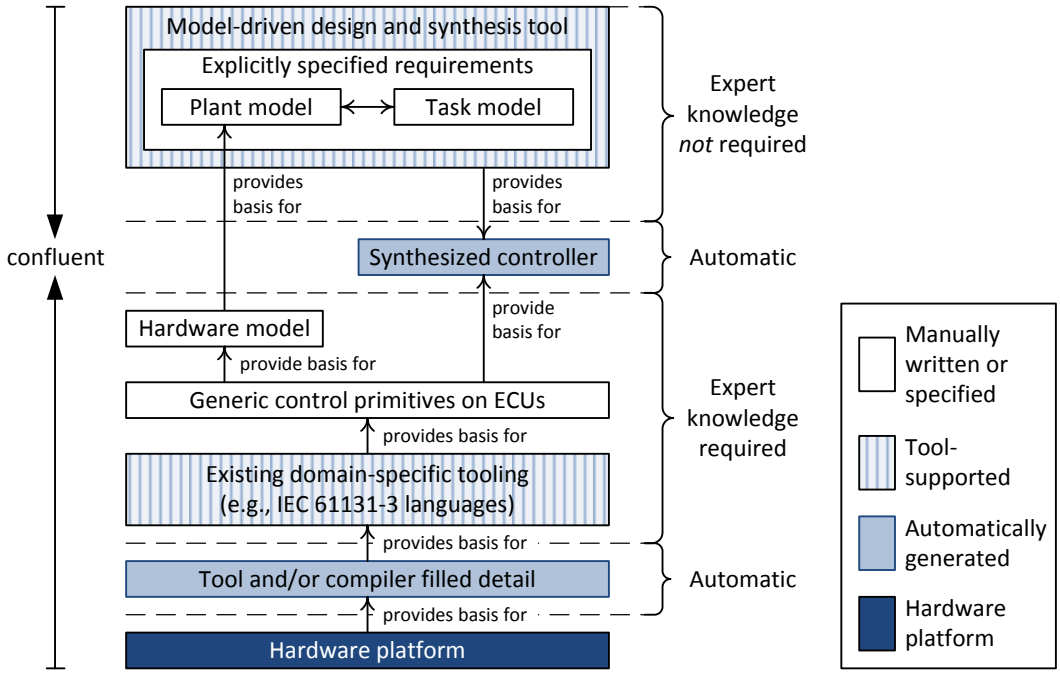


Figure 3.3.: Proposed workflow for control program synthesis: just like in the traditional approach, primitive function blocks are created using domain specific tooling. However, the behavior is then encapsulated in form of a hardware model that is used by non-experts to synthesize the actual control programs from a plant model and a task model. Those models explicitly represent the respective requirements. This approach is further referred to as a *confluent* workflow.

3.2.2.2. Hardware Model, Plant Model and Task Specification

Based on the control primitives, a *hardware model* is created by experts in mechatronics that lists for every hardware module its interfaces along with conditions when it is legal to trigger them. The hardware model also includes the individual states in which a hardware module can be.

The hardware model is input into a model-driven development and synthesis tool that serves as a front-end to allow specification of the actual plant setup (the so-called *plant model*) and the control task (*task model*).

3.2.2.3. Synthesis

Both the models from the domain expert as well as from the non-expert are used to synthesize the concrete (decentralized) control programs for the automation system. This automatic step reduces the potential for mistakes.

Synthesis is performed using a game-based solver that is attached to the modeling tool and derives a concrete implementation from the given models. If a solution exists, the output of the solver is a so-called *strategy*, i.e., a sequential program where each action is guarded by a condition that ensures that an action is only executed when it is legal to do so. If multiple solutions exist for a given specification, a cost model as specified in **Goal 3** is used to decide which implementation to choose.

The selected solution can be simulated on the development machine or executed on the real hardware. For this purpose, code is generated for the respective target platforms using a template-based code generator. Where necessary, the templates are based on a platform abstraction layer in order to support execution on different target platforms (**Goal 5**). In case of distributed target systems, the necessary configuration and communication patterns are also generated, satisfying **Goal 4**.

As indicated by the arrows on the left of Figure 3.3, this workflow is partially *bottom-up*, where higher-level system components are based on low-level building blocks, and partially *top-down*, where the system is first specified in a coarse-grained way and then (automatically) refined. Hence, we call this approach a *confluent* workflow.

3.2.3. Relation to the Automation Pyramid

The presented approach is related to the automation pyramid introduced in Figure 2.2 on page 14 as follows: this work provides a model-driven development tool at *area level* (level 3) that allows to describe key properties of the elements at *control*, *field*, *process* and *supervisory level* (levels 0 to 2). Furthermore, the development tool allows the specification of an abstract production goal. Using the synthesis engine, the tool generates a plan (*planning*) that satisfies the production goal and guarantees that given cost bounds as not exceeded (*optimization*). The generated plan is subsequently transformed into a decentralized control strategy (*control*) for the individual control units at the *control level* (level 1).

3.3. Scope of this Work

3.3.1. Considered Industrial Automation Systems

A study in the United Kingdom from 2004 [Bak04] concludes that automation is common "in large warehouses, particularly with regard to conveyor/sortation, and automated storage and retrieval systems, with each of these types of equipment being present in more than a third of large warehouses" [BH07]. One reason for this is that the processes carried out here are comparably easy to automate, because their primary intent is to move goods on rather fixed paths. This also leads to an increased level of comfort for human workers. Such systems are subject to research in this work.

More precisely, this work assumes that the industrial automation systems under consideration are automation lines that consist of a number of discrete mechatronic hardware *modules*. A module has the ability to process (e.g., transport, alter, store) *work pieces* (which represent an arbitrary product), more precisely it has a set of *operating positions* on which the physical state of one or multiple work pieces may be altered. Work pieces are considered to be solid; "bulk" materials (such as granulate or fluids) are not taken into account.

Furthermore, a module may have an arbitrary amount of *sensors* and *actuators*. Sensors make information from the plant's environment accessible to the control algorithm (e.g., a light barrier detects whether a work piece is currently present at a specific operating position). Actuators allow the control algorithm to change the state of the environment in a distinct manner (e.g., a conveyor belt can transfer work pieces from one operating position to another).

This work refers to the described class of automation systems as *modular assembly lines* (MALs).

3.3.2. Discrete Modeling

In this work, a game-based solver is used to synthesize control programs for industrial automation. The solver is currently only capable of handling a discrete state space. This means that modeling of equipment with continuous state (e.g., industrial robots with a number of degrees of freedom) requires some additional work: the positional state space of such a robot has to be *discretized* in order to fit our modeling scheme. Discretization means that the possible positional space defined by the robot is reduced to a finite number of discrete operating positions that the robot can reach. Notice that this work does *not* deal with robot control theory as such, namely kinematics or motion planning. In this work, it is assumed that a suitable control algorithm exists that allows to position the robot according to the given input parameters.

The rationale for the restriction to discrete domains is that the used solver is optimized for discrete problems. Although continuous ranges of values could be encoded in the input language of the solver, it would drastically increase the synthesis time due to the complexity of the problem. Hence, we assume it to be the responsibility of the control engineer to pick a meaningful discretization for continuous values.

3.3.3. No Control-theoretic Algorithm Synthesis

This work is *not* about synthesizing a control algorithm in the context of control theory. This work focuses on synthesis of a strategy that triggers primitive actions in a way such that a specified state in discrete state space is reached. Although a looped execution of a generated control program could be used to reach and hold a specific control state, such tasks are not in the focus of this work. Any required control-theoretic algorithms (e.g., *proportional-integral-derivative (PID)* controllers) are assumed to be implemented in *primitive control functions (PCFs)* that are parameterized or triggered by the synthesized control strategy. Many specialized tools for effective design of low-level control algorithms exist (e.g., [The13]) that can be used in conjunction with the approach presented in this thesis.

3.3.4. Target Platforms

The presented toolchain includes code generation for execution "on the real hardware" as well as simulation. Code generation supports the following target platforms:

- **(Industrial) *personal computers (PCs)* with x86 architecture and Windows or Linux operating system:** These types of computers are widely used in industrial context for monitoring and control in form of PCs or panel computers.
- **Embedded systems with ARM microcontrollers:** This family of systems represents smart sensors and actuators with "local intelligence". Those systems have certain calculating and storage capabilities.

Target platforms that are *not* taken into account are:

- **Programmable Logic Controllers (PLCs):** Although possible, code generation for PLCs is not a goal of this thesis. One of the reasons for this is that existing tools (as presented in Section 2.1) handle modeling and – to some extent – code generation in this area. Instead of generating code for PLCs, the tools created along with this work allow to interact with existing control programs on a PLC.

Simulation supports the following concepts:

- **Generated text mode simulator:** The presented toolchain allows generation of a simulator application for every feasible task. The generated simulator runs in text mode (i.e., it is a console application) and allows the user to specify input values from the environment that would be captured by sensors in a real setup.
- **OPC connection:** An *open platform communications (OPC)* [IL01] interface allows the generated text mode simulator to control OPC-based simulation software. For example, a 3D model in CIROS Studio [RIF13, HH12] can be controlled using this setup. In this case, sensor readings are also obtained from the 3D simulation. This approach provides a meaningful visualization and feedback for industrial automation systems[1] that should be simulated before actually building them or to test certain concepts in hypothetical automation lines.

[1] It should be mentioned that the time required to set up the configuration of, for example, CIROS Studio to simulate the automation system correctly is not negligible. This process requires expert knowledge in 3D simulation and automation.

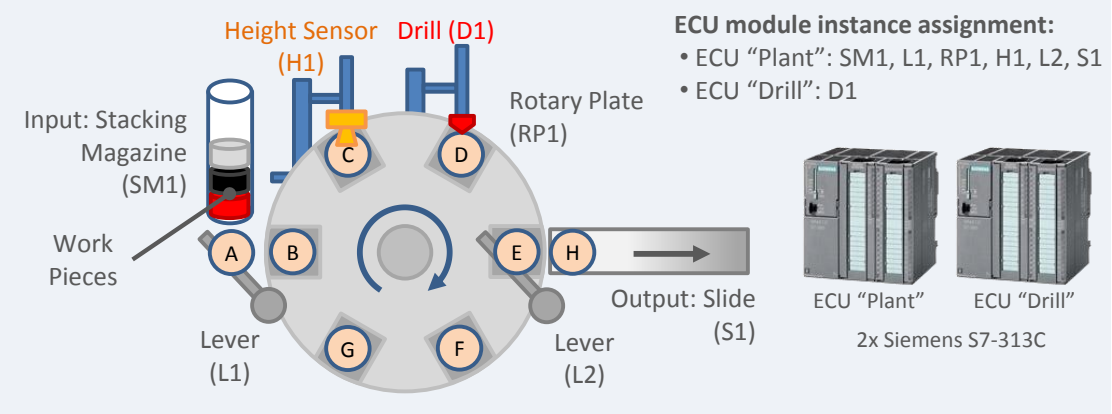


Figure 3.4.: Running example: work pieces that are initially located in a *stacking magazine* are separated by a *lever*. On a *rotary plate* with six operating positions, work pieces may be transported underneath a *height sensor* that detects whether a work piece has a hole or not. Furthermore, work pieces may be put underneath a drill for drilling. Finally, another *lever* pushes work pieces onto a *slide* that serves as output (ECU icon by [Sie14a]).

3.3.5. Communication

When the control algorithm for an automation system is to be distributed among multiple ECUs, a control strategy is generated for each individual ECU based on the global control strategy. In order to achieve the "global goal", messages typically need to be exchanged between ECUs. In this work, we assume that all ECUs are able to communicate with each other.

The presented algorithm generates ECU-local control programs that run independent of each other until synchronization is (potentially) required to exchange information on the current state. Since the fact whether the state exchange is necessary or not may depend on input from the environment and each ECU has its own limited view on the system state, a safe over-approximation on when synchronization is necessary is used. More precisely, a synchronization operation is introduced between a pair of ECU instances $\hat{u}_1$ and $\hat{u}_2$ in execution step n if a state exchange is required in any of the possible execution traces in the global control strategy in step n.

In this approach, the granularity of synchronization is always at the scope of Controller or Environment "moves" (i.e., synchronization is only performed in between two control moves of an ECU).

3.4. Introduction of the Running Example

In the following, an example MAL is introduced that is used for illustration purposes within the next three chapters. Notice that the running example is kept simple on purpose. Please refer to Chapter 7 for more complex examples.

Consider the modular assembly line depicted in Figure 3.4, which consists of the following hardware modules:

- **Stacking magazine SM1:** A magazine that holds a number of stacked work pieces for processing in the plant. The work piece on the bottom is located at operating position A. The magazine serves as input for work pieces into the plant.
- **Lever L1:** This lever pushes a work piece at position A to position B (on the rotary plate).
- **Rotary plate RP1:** The rotary plate offers six positions that can hold one work piece each. The plate rotates in clockwise direction only. In one rotation step of 60°, the plate simultaneously moves work pieces by one position (e.g., a work piece on position B is moved to position C, a work piece on position G is moved to position B).
- **Height sensor H1:** This module detects whether a work piece at position C has a hole or not. It consists of an electromagnet that, when supplied with power, presses a rod against the work piece. Whether the rod penetrates the work piece (and hence the work piece has a hole) is tested by an electromagnetic sensor mounted next to the rod.
- **Drill D1:** This module drills a hole into a work piece located at position D. It consists of a linear axis that moves up and down. The drill is mounted on the linear axis. Drilling a work piece consists of the following steps: switching on the drill, moving down the drill on the linear axis, moving up and switching off the drill. In order to determine whether the drill has reached its intended position on the linear axis, digital input signals are provided on both ends.
- **Lever L2:** This lever is able to push a work piece at position E to position H on the slide. It is mounted in a way such that a work piece may still be moved from position E to position F when the rotary plate rotates.
- **Slide S1:** After a work piece is moved to position H on the slide, it drops towards the end of the slide (on the right side of Figure 3.4) by the force of gravity. The slide buffers a number of work pieces and serves as output from the plant.

In this example, two ECUs of type Siemens S7-313C are used to control the sensors and actuators. ECU "Plant" controls all modules except for the drill, which is controlled by ECU "Drill". Assume that the ECUs are attached to a *Multi-Point Interface (MPI)* bus and configured as the devices with physical MPI address 1 and 2.

In this example, assume the following simplifications:

- When a task specification denotes the presence of a work piece at input position A, we assume that such a work piece is present at that position. Hence, it is expected that the state of the plant is consistent with the initial condition specified in the model.
- When a task specification denotes the output of a work piece on position H, we assume that enough space is available on the slide for the work piece to leave position H by the force of gravity. Furthermore, it is assumed that this happens instantaneously.

The following trivial automation tasks may be carried out in this plant, for example:

- Transfer of work pieces from the stacking magazine to the slide.
- Detection of the drilled state of each work piece and output on the slide.
- Unconditional drilling of each work piece and output on the slide.
- Drilling of undrilled work pieces and output on the slide.
- Simultaneous buffering of up to six work pieces on the rotary plate and subsequent output on the slide.

This running example will be used throughout the next three chapters to illustrate how the formal model captures the properties of MALs.

CHAPTER 4

System Modeling and Task Description

Contents

4.1. Modeling Overview 44
4.2. Formal Description of Modular Assembly Lines 44
4.3. Formal Description of Automation Tasks 53
4.4. Discussion and Application to Running Example 56
4.5. Summary 70
4.6. Related Work 71

Overview

This chapter answers the following research question:

> How can modular assembly lines and the tasks to be performed on them be formally described such that the description is applicable to automated processing, such as control software synthesis?

This question is answered by defining a domain-specific language at an appropriate level of abstraction. Aspects such as the hardware modules and *electronic control units (ECUs)* in the assembly line, a discrete representation of the plant's state and a formalization of the actions that can be performed form the base of this language. In addition, a formal language for task description is required for control software synthesis. The specification of the plant and the task description are illustrated in conjunction with the running example.

4.1. Modeling Overview

Section 3.3.1 defined the types of automation systems that are considered in this work, namely so-called *modular assembly line (MAL)*. To summarize, a MAL consists of a topological arrangement of hardware components (called *modules*), where each module is designed to handle work pieces on a set of operating positions. Hence, a *(hardware) module* is a combination of sensors and/or actuators with a non-empty set of capabilities as well as passive elements (such as mechanical parts).

The running example introduced in Section 3.4 contains various such hardware modules, namely a stacking magazine, two levers, a rotary plate, a height sensor module, a drill module and a slide.

In this section, a *metamodel* for MALs and their respective hardware modules is introduced. The definition of the formalism consists of three steps:

1. **Hardware model:** The first step is to define a formalism with which experts in mechatronics can describe the structure, behavior, state, *input/output (I/O)* interfaces and configurable parameters of individual MAL modules and the respective ECUs.
2. **Plant model:** The second step is to define a formalism with which experts in automation can topologically arrange a number of modules to form a concrete automation system. The plant model contains concrete module and ECU instances.
3. **Task model:** The third step is to define a formalism for specifying a task to execute on the concrete automation system. The task specification is based on logic formulas over the state space of the MAL. In principle, this step does not need to be performed by an expert in industrial automation.

The following sections define requirements for the aspects that need to be represented in the model and derive a *domain-specific language (DSL)* that satisfies the requirements. The notation "$\models \mathrm{R}x$" is used to indicate that a concept fulfills requirement x.

4.2. Formal Description of Modular Assembly Lines

This section presents the approach used to formally describe MAL modules. First, a list of requirements for a suitable metamodel is provided. The requirements are designed in a way that an algorithm can automatically synthesize control software from a concrete model conforming to the metamodel.

4.2.1. Requirements

For modeling MALs, the modules they consist of need to be represented.

Requirement R1 (module types) *The individual components of MALs need to be represented in form of* module *types. A module type is a meaningful combination of, among others, mechatronic, electric, pneumatic and hydraulic components and may include sensors and actuators.*

An example for a module type is the drill from Section 3.4. It consists of a linear axis with an electric motor and the drill itself as actuators as well as sensors to detect the position of the drill on the linear axis. The next requirement relates module types to positions where work pieces can be located.

Requirement R2 (operating positions) [1]*For every module type, the set of associated* operating positions *need to be specified. An operating position is a discrete location where a work piece may be located.* [2]*If operating positions are physically incompatible with each other (e.g., some operating positions handle different work pieces than others), they have to be separated into respective groups.*

The drill's single operating position is the location where a work piece can be drilled. A real plant consists of instances of module types. Multiple instances of the same module type can be incorporated, such as a number of drill modules.

Requirement R3 (module instances) *The model must capture the fact that a plant consists of an arbitrary amount of module instances from the different module types.*

In order to build a processing chain, neighboring module instances must be able to exchange work pieces. For this purpose, the concept of overlapping operating positions is defined:

Requirement R4 (overlapping operating positions) *The model must be capable of expressing that two or more operating positions of different module instances physically overlap each other, i.e., they represent the same physical space in the plant.*

This mechanism ensures that work pieces can be exchanged between module instances. In the running example from Section 3.4, the drill's operating position overlaps one of the operating positions of the rotary plate.

Next, the fact that every module instance has a configuration (or state) that evolves during runtime needs to be represented. For this purpose, assume that the subset of the state that is relevant for the automation task is discretized and represented in the formal model.

Requirement R5 (module state) [1]*For every module type, the relevant* state space *needs to be specified in a discrete way.* [2]*The initial state of all module instances upon start-up needs to be specified as well.* [3]*The state must be adaptable at runtime in order to represent the evolution of the production task.* [4]*A module type may also have some additional configuration parameters that influence its behavior.*

In order to control the production process, the next step is to formally define the capabilities of each module type. For example, the drill module's most relevant capability is to drill a work piece. Capabilities are represented as actions with preconditions and effects on the state of the respective module instance.

Requirement R6 (actions) [1]*For every module type, the primitive* actions *that it can perform need to be specified (e.g., processing of a work piece, measurement of a physical value).* [2]*The effects of actions should be limited to the operating positions of the respective module type.* [3]*Each action needs to be guarded with preconditions on the state of the respective module instance and the environment (i.e., the preconditions need to hold in order for the action to be executable). The execution of an action has two effects:* [4]*on the one hand, the respective control action is triggered in the "real" plant.* [5]*On the other hand, the representation of the state of the module instance and the environment may change in order to capture the effects the control action has in the "real" plant.* [6]*For quantitative evaluation and optimization, the cost of each action (in terms of a respective cost model) needs to be specified.*

In order to obtain a continuous development process, we need to be able to automatically map synthesized control programs for the ECUs.

Requirement R7 (hardware mapping) *The model needs to express all required parameters in order to map the actions of the module types to the respective hardware platform; this includes the set of supported ECU types and information about the required toolchain.*

Apart from the state of the module instances mentioned in R5, we also need to represent the state of other objects in the plant, for example the state of the work pieces.

Requirement R8 (plant state) [1]*Objects in the plant (e.g., work pieces) need to be formally represented.* [2]*If objects are of different type, they have to be separated into respective groups.* [3]*For every object in the plant, the relevant state needs to be specified in a discrete way.* [4]*The state must be adaptable at runtime in order to represent the evolution of the production task.*

For example, the drilled state needs to be represented for every work piece. ECU instances are connected to the individual module instances in order to control them. The association between ECU instances and module instances needs to be known for automatic control software synthesis.

Requirement R9 (ECUs and communication infrastructure) [1]*It must be specified which ECU instances control which module instances.* [2]*It must be known which I/O signals are used to trigger the respective actions in the target hardware.* [3]*In addition, it must be specified how every ECU instance is connected to the communication network.*

4.2.2. Formalism Implementing the Requirements

This section introduces a formalism that implements all listed requirements. First, some basic concepts are introduced that are required as a base for the following sections. Subsequently, a formal description of hardware modules of a MAL and their capabilities is given. Finally, a description is presented that allows specifying concrete MALs built from those hardware modules.

4.2.2.1. Basic Concepts

Definition D1 (object, object type) *Objects are constants that represent* work pieces*, literal properties of work pieces,* operating position *instances of module instances and properties of the environment (⊨ R8.1). Formally, the set of all* objects $\mathcal{X}$ *is partitioned into disjoint sets of* object types $\mathcal{X}_0, \mathcal{X}_1, \ldots \mathcal{X}_{\mathtt{t}-1}$ *with* $t \geq 1$ *(⊨ R2.2, ⊨ R8.2) such that:*

$$\textit{Collective exhaustion:} \quad \bigcup_{0 \leq i < \mathtt{t}} \mathcal{X}_i = \mathcal{X} \tag{4.1}$$

$$\textit{Mutual exclusion:} \quad \forall 0 \leq i, j < \mathtt{t} \colon \mathcal{X}_i \cap \mathcal{X}_j = \emptyset \vee i = j \tag{4.2}$$

For example, the set of objects $\mathcal{X}$ for representing days of the week may be partitioned into the object types $\mathcal{X}_{\mathsf{weekday}} = \{x_{\mathsf{Mon}}, x_{\mathsf{Tue}}, x_{\mathsf{Wed}}, x_{\mathsf{Thu}}, x_{\mathsf{Fri}}\}$ and $\mathcal{X}_{\mathsf{weekend}} = \{x_{\mathsf{Sat}}, x_{\mathsf{Sun}}\}$ with $\mathtt{t} = 2$ and $\mathcal{X} = \mathcal{X}_{\mathsf{weekday}} \cup \mathcal{X}_{\mathsf{weekend}}$.

Definition D2 (predicate) *Predicates represent the state of* modules *and* objects *(required for R5 and R8). Formally, a* predicate $v \in \mathcal{V}$ *is a function*

$$v \colon \mathcal{X}_{v_0} \times \mathcal{X}_{v_1} \times \cdots \times \mathcal{X}_{v_{a-1}} \to \mathbb{B} \tag{4.3}$$

where:

- $\mathcal{X}_{v_i}, 0 \leq i < a$ *are the arguments of the predicate where* $\mathcal{X}_{v_i} = \bigcup_{j \in \mathtt{t}_v(v,i)} \mathcal{X}_j$ *is the set of all objects allowed for the respective argument, defined by the union of the allowed object types according to the function* $\mathtt{t}_v(v, i)$*.*
- $a \geq 0$ *is the number of arguments of the predicate.*
- $\mathbb{B} = \{0, 1\}$ *is the set of Boolean values.*

For example, the zero-argument predicate "sun-shining" indicates whether the sun is currently shining and the one-argument predicate "sun-shining-on($\mathsf{a}_{\mathsf{day}}$)" with $\mathtt{t}_v(\mathsf{sun\text{-}shining}, 0) = \{\mathsf{weekday}, \mathsf{weekend}\}$ indicates whether the sun is shining on the given day of the current week.

Definition D3 (condition) *A* condition $c \in \mathcal{C}$ *is an assignment of all arguments of a predicate* $v \in \mathcal{V}$ *with matching objects (according to the expected type of the respective argument; required for R6)*

$$c \equiv v(\mathsf{a}_{v_0}, \mathsf{a}_{v_1}, \ldots, \mathsf{a}_{v_{a-1}}) \tag{4.4}$$

where a_{vi} *is the* i^{th} *item in the ordered list of objects passed to the arguments of predicate* v *with*

$$\forall 0 \leq i < a \colon \mathsf{a}_{vi} \in \mathcal{X}_{v_i} \tag{4.5}$$

Examples:

- The condition sun-shining is true if the sun is currently shining.
- The condition sun-shining-on(x_{Tue}) is true if the sun is shining on Tuesday.

4.2.2.2. Formal Description of Hardware Modules and Capabilities

In the following, a mathematical formalism for the specification of MAL modules and their capabilities is introduced. The purpose of these definitions is to serve as a base for deriving a suitable domain specific language. Notice that these definitions are provided with almost no concrete examples. Please refer to Section 4.4.1 for respective illustrations in the context of the running example.

Definition D4 (hardware model) *A* hardware model $\mathfrak{H}$ *is a tuple*

$$\mathfrak{H} = (\Phi_{\mathfrak{H}}, \mathcal{V}_{\mathfrak{H}}, \mathcal{X}_{\mathfrak{H}}, \mathcal{U}_{\mathfrak{H}}) \tag{4.6}$$

where:

- $\Phi_{\mathfrak{H}} \subseteq \Phi$ *is the set of* module types*. A module type formally represents the properties and capabilities of a specific hardware module. See Definition D5 for a formal definition of module types.*
- $\mathcal{V}_{\mathfrak{H}} \subseteq \mathcal{V}$ *is the set of* predicates *available in the hardware model (⊨ R5.1, ⊨ R8.3).*
- $\mathcal{X}_{\mathfrak{H}} \subseteq \mathcal{X}$ *is the set of* objects *available in the hardware model (⊨ R8.1). The objects in this set serve as constants for predicate evaluation. In the example for Definition D1, the individual days* $x_{\mathsf{Mon}}, \ldots, x_{\mathsf{Sun}}$ *would be represented as objects in this set.*
- $\mathcal{U}_{\mathfrak{H}} \subseteq \mathcal{U}$ *is the set of* ECU types *available in the hardware model. An ECU type is a formal representation of key properties of an ECU. See Definition D9 for a formal definition of ECU types.*

Definition D5 (module type) *A* module type $\varphi \in \Phi$ *(⊨ R1) is a tuple*

$$\varphi = (\mathcal{B}_{\varphi}, \Pi_{\varphi}, \mathcal{I}_{\varphi}, \mathcal{O}_{\varphi}, \mathcal{P}_{\varphi}) \tag{4.7}$$

where:

- $\mathcal{B}_{\varphi} \subseteq \mathcal{B}$ *is the set of* behavioral interfaces*. A behavioral interface is the formalization of a primitive action that the associated module can perform (⊨ R6.1, see Definition D6 for a formal definition). A module type* φ *with* $\mathcal{B}_{\varphi} = \emptyset$ *is called* passive*. All module types that are not passive are called* active*.*
- $\Pi_{\varphi} \subseteq \mathcal{X}$ *with* $\Pi_{\varphi} \neq \emptyset$ *is the set of* operating positions *(⊨ R2.1). An operating position is a position on the module that can be occupied by a work piece for processing.*
- $\mathcal{I}_{\varphi} = \{\iota_{\varphi_0}, \iota_{\varphi_1}, \ldots, \iota_{\varphi_{\iota-1}}\}$ *is the set of* input signals *(⊨ R9.2). An input signal is used to make the control program aware of environmental conditions (e.g., a digital signal from a light barrier indicates whether a work piece is in front of it or not).*
- $\mathcal{O}_{\varphi} = \{o_{\varphi_0}, o_{\varphi_1}, \ldots, o_{\varphi_{o-1}}\}$ *is the set of* output signals *(⊨ R9.2). An output signal is used to allow the control program to trigger actions in the process environment (e.g., a digital signal for switching on or off the motor of a conveyor belt).*
- $\mathcal{P}_{\varphi} = \{p_{\varphi_0}, p_{\varphi_1}, \ldots, p_{\varphi_{p-1}}\}$ *is the set of* configurable module type parameters *that specify degrees of freedom in the configuration of the module type. They allow a developer to fine-tune the behavior of the respective module instance (⊨ R5.4, see Definition D12).*

Definition D6 (behavioral interface) *A behavioral interface is an action that may be triggered when certain preconditions are met. Upon triggering of a behavioral interface, the state of the system, represented by the values of the predicates in the respective hardware model, may change.*

Formally, a behavioral interface $\mathsf{b} \in \mathcal{B}$ *is a tuple*

$$\mathsf{b} = (\mathsf{A}_\mathsf{b}, \mathcal{C}_{\mathcal{P}\mathsf{b}}, \mathcal{E}_\mathsf{b}, \widehat{\mathcal{E}}_\mathsf{b}, \mathcal{R}_\mathsf{b}, \hat{p}_\mathsf{b}, \eta_\mathsf{b}, \ddot{p}_\mathsf{b}) \tag{4.8}$$

where:

- $\mathsf{A}_\mathsf{b} = (\mathcal{X}_{\mathsf{b}_0}, \mathcal{X}_{\mathsf{b}_1}, \ldots, \mathcal{X}_{\mathsf{b}_{b-1}})$ *is the ordered list of* argument domains *of each of the parameters of the behavioral interface. All objects and operating positions that are affected by this behavioral interface need to be represented by a parameter and hence included in the list of argument domains (⊨ R6.2).*
- $\mathcal{C}_{\mathcal{P}\mathsf{b}}$ *is the set of* behavioral interface preconditions *(⊨ R6.3). A* precondition $c_p \in \mathcal{C}_{\mathcal{P}\mathsf{b}}$ *conforms to the concept of conditions as specified in Definition D3 with the exception that the parameters assigned to the arguments of the predicate in the condition may be references to matching arguments of the containing behavioral interface* b*, that is (compare equation (4.4)):*

  $$c_p \equiv v(\mathsf{a}_{v_0}, \mathsf{a}_{v_1}, \ldots, \mathsf{a}_{v_{a-1}}) \tag{4.9}$$

  with (compare equation (4.5))

  $$\forall 0 \leq i < a \colon \mathsf{a}_{v_i} \in \mathcal{X}_{v_i} \cup \{\mathsf{A}_{\mathsf{b}_i} \mid \mathcal{X}_{\mathsf{b}_i} = \mathcal{X}_{v_i}\} \tag{4.10}$$

  Furthermore, preconditions may be negated by prepending "¬". All preconditions of a behavioral interface need to evaluate to the Boolean value 1 *(true) for the behavioral interface to be executable.*

  Examples:

  - The behavioral interface precondition $\mathsf{sun\text{-}shining\text{-}on}(x_{\mathsf{Tue}})$ allows execution of the behavioral interface if the sun is shining on Tuesday.
  - The behavioral interface precondition $\neg\mathsf{sun\text{-}shining\text{-}on}(x_{\mathsf{Tue}})$ allows execution of the behavioral interface if the sun is *not* shining on Tuesday.
  - The behavioral interface precondition $\mathsf{sun\text{-}shining\text{-}on}(\mathsf{a}_{\mathsf{b}_0})$ allows execution of the behavioral interface depending on whether the sun is shining on the day specified by the parameter $\mathsf{a}_{\mathsf{b}_0}$ of behavioral interface b. In this case, the parameter list of b must contain a respective matching argument, for example $\mathsf{A}_\mathsf{b} = (\mathcal{X}_{\mathsf{b}_0})$ with $\mathcal{X}_{\mathsf{b}_0} = \mathcal{X}_{\mathsf{weekday}} \cup \mathcal{X}_{\mathsf{weekend}}$.
- $\mathcal{E}_\mathsf{b}$ *is the ordered list of* unconditional behavioral interface effects *(⊨ R6.5). An* unconditional effect $e \in \mathcal{E}_\mathsf{b}$ *is the assignment of a Boolean value to a predicate* $v \in \mathcal{V}_{\mathfrak{H}}$ *using valid parameters for the arguments of the predicate. After the behavioral interface is executed, the affected predicates' values at the given parameters are set to the respective Boolean values (⊨ R5.3, ⊨ R8.4). The unconditional effects may reference arguments from the parent behavioral interface* b *in order to set a predicate according to a value passed to the behavioral interface. In the following,* e *is written as*

"$\neg v(x_{v_0}, x_{v_1}, \ldots x_{v_{a-1}})$" to indicate that the respective Boolean value is set to false and is it written as "$v(x_{v_0}, x_{v_1}, \ldots x_{v_{a-1}})$" to indicate that it is set to true.

Examples:

- The unconditional behavioral interface effect is-dry states that it is dry (i.e., no rain). It sets the zero-argument predicate is-dry to true. Similarly, the unconditional behavioral interface effect ¬is-cloudy sets the zero-argument predicate is-cloudy to false. Both are meaningful in combination with precondition sun-shining, for example.
- The unconditional behavioral interface effect is-dry-on(x_{Tue}) states that it is dry on Tuesday and ¬is-dry-on(x_{Tue}) implies that it is not dry on Tuesday.
- The unconditional behavioral interface effect is-dry-on($\mathsf{a}_{\mathsf{b}_0}$) states that it is dry on the day specified by the parameter $\mathsf{a}_{\mathsf{b}_0}$ of behavioral interface b. This is meaningful in combination with precondition sun-shining-on($\mathsf{a}_{\mathsf{b}_0}$). In this case, the parameter list of b must contain a respective matching argument, for example $\mathsf{A}_\mathsf{b} = (\mathcal{X}_{\mathsf{b}_0})$ with $\mathcal{X}_{\mathsf{b}_0} = \mathcal{X}_{\text{weekday}} \cup \mathcal{X}_{\text{weekend}}$.

- $\widehat{\mathcal{E}}_\mathsf{b}$ *is the ordered list of* conditional behavioral interface effects *(⊨ R6.5). A* conditional effect $\hat{e} \in \widehat{\mathcal{E}}_\mathsf{b}$ *is a pair in which the first element specifies a set of conditions $\mathcal{C}_{\mathcal{P}\mathsf{b}_{\hat{e}}}$ (conforming to the specification of preconditions above) and the second element specifies a set of effects that is applied when all conditions are true at the point in time when the containing behavioral interface* b *has finished executing. After the behavioral interface is executed, the affected predicates' values at the given parameters are only set to the respective Boolean values*[1] *if the additional precondition evaluates to the Boolean value* 1 *(⊨ R5.3, ⊨ R8.4). Both conditions and effects may reference arguments from the parent behavioral interface* b *in order to test respectively set a predicate according to a value passed to the behavioral interface.*
- $\mathcal{R}_\mathsf{b}$ *is the set of* sensor result conditions*. A sensor result condition $r \in \mathcal{R}_\mathsf{b}$ is a tuple (r_ρ, r_e), where $r_\rho \in \mathbb{Z}$ is the literal sensor response value as it is returned by the underlying platform*[2] *and r_e is an* unconditional effect *that is applied when a sensor triggering returns the respective literal sensor value.*
- $\hat{p}_\mathsf{b}$ *is the name of the* primitive control function *on the target system to invoke in order to trigger the primitive control program that implements the behavioral interface (⊨ R6.4). The necessary parameters are passed to the function as explained later in Section 6.3.1.*
- $\eta_\mathsf{b} \in \mathbb{N}_0$ *is the* cost *of the execution of the behavioral interface with respect to the used cost model (⊨ R6.6, specified later in Definition D14 in Section 4.3.2).*
- $\ddot{p}_\mathsf{b} \in \mathbb{B}$ *is a flag indicating whether this behavioral interface should be considered for parallelization. Due to limitations in the solver, some behavioral interfaces with conditional effects are incompatible with parallelization. This flag can be used to deactivate parallelization for them if necessary. The default value is 1 (i.e., parallelization is enabled).*

[1] Without loss of generality, if unconditional and conditional effects are both present, the conditional effects are applied after the unconditional effects.

[2] It is assumed that the possible sensor responses are mapped to the discrete domain $\mathbb{Z}$.

Definition D7 (actuation) *A behavioral interface* $\mathsf{b} \in \mathcal{B}$ *with no sensor result conditions, i.e.,* $\mathcal{R}_\mathsf{b} = \emptyset$*, is called* actuation.

Definition D8 (sensor triggering) *A behavioral interface* $\mathsf{b} \in \mathcal{B}$ *with a non-empty set of sensor result conditions, i.e.,* $\mathcal{R}_\mathsf{b} \neq \emptyset$*, is called* sensor triggering.

Definition D9 (ECU type) *An* ECU type $u \in \mathcal{U}$ *is a tuple*

$$u = (\xi_u, \mathcal{I}_u, \mathcal{O}_u, \mathcal{P}_u) \tag{4.11}$$

where:

- $\xi_u \in \Xi$ *specifies the target platform of the ECU (⊨ R7). This information is required for determining how to program and/or control the ECU (e.g., a* programmable logic controller (PLC) *is remotely controlled while code is generated for a microcontroller). It is assumed that this element references information such as the (compiler) toolchain and settings required for generating and compiling code for the respective platform as well as all dependencies and third-party tools that are required in this process.*
- $\mathcal{I}_u = \{\iota_{u_0}, \iota_{u_1}, \ldots, \iota_{u_{\iota-1}}\}$ *is the set of all* input channels *offered by this ECU type (⊨ R9.2).*
- $\mathcal{O}_u = \{o_{u_0}, o_{u_1}, \ldots, o_{u_{o-1}}\}$ *is the set of all* output channels *offered by this ECU type (⊨ R9.2).*
- $\mathcal{P}_u = \{p_{u_0}, p_{u_0}, \ldots, p_{u_{p-1}}\}$ *is the set of* configurable parameters *of the ECU type. Typical properties represented here are the address of the ECU on a given communication bus for remote control as well as inter-ECU communication when using decentralized control programs (⊨ R9.3, covered later in Section 6.4).*

4.2.2.3. Formal Description of Concrete Modular Assembly Lines

The formalism introduced in the previous section allows experts in mechatronics to define a library of module types along with their properties and behavioral interfaces in form of a hardware model. Based on a given hardware model, experts in industrial automation define the model for a concrete modular assembly line by instantiating modules and parameterizing them accordingly.

Notice that the following definitions are provided without concrete examples. Please refer to Section 4.4.2 for respective illustrations in the context of the running example.

Definition D10 (plant model) *A* plant model $\mathfrak{P}$ *is a tuple*

$$\mathfrak{P} = (\mathfrak{H}_\mathfrak{P}, \mathcal{X}_\mathfrak{P}, \Psi_\mathfrak{P}, \widehat{\mathcal{U}}_\mathfrak{P}, \mathsf{L}_\mathfrak{P}) \tag{4.12}$$

where:

- $\mathfrak{H}_\mathfrak{P}$ *is the* hardware model *referenced by this plant model.*
- $\mathcal{X}_\mathfrak{P} \subseteq \mathcal{X}$ *is the set of* objects *available in the plant model. This set complements the set of objects* $\mathcal{X}_\mathfrak{H}$ *in the hardware model by declaring operating position instances and work piece instances.*

- $\Psi_{\mathfrak{P}} \in \Psi$ *is the set of* module instances *(⊨ R3). See Definition D12 for a formal definition of module instances.*
- $\widehat{\mathcal{U}}_{\mathfrak{P}} \in \widehat{\mathcal{U}}$ *is the set of* ECU instances *(⊨ R9.1). See Definition D11 for a formal definition of ECU instances.*
- $\mathsf{L}_{\mathfrak{P}}$ *is the list of* overlapping operating positions *(⊨ R4). See Definition D13 for a formal definition of overlapping operating positions.*

Definition D11 (ECU instance) *An* ECU instance $\hat{u} \in \widehat{\mathcal{U}}$ *is a tuple*

$$\hat{u} = (u_{\hat{u}}, \vec{u}_{\hat{u}}) \tag{4.13}$$

where:

- $u_{\hat{u}} \in \mathcal{U}$ *denotes the* ECU type *of this ECU instance.*
- $\vec{u}_{\hat{u}} \colon \mathcal{P}_{u_{\hat{u}}} \to \Sigma^*$ *is the* ECU instance parameter assignment function *that maps each of the configurable parameters of the respective ECU type $u_{\hat{u}}$ to a specific value, where*

$$\Sigma^* = \bigcup_{n \in \mathbb{N}_0} \Sigma^n \tag{4.14}$$

is the set of character strings with finite length (including the empty string $\epsilon = \Sigma^0$) over the alphabet Σ, which is assumed to be the set of printable characters from the American Standard Code for Information Interchange (ASCII) *[IET68] character set.*

Definition D12 (module instance) *A* module instance $\psi \in \Psi$ *is a tuple*

$$\psi = (\varphi_\psi, u_\psi, \Theta_\psi, \vec{\theta}_\psi, \mathcal{C}_{\mathcal{I}\psi}, \vec{\iota}_\psi, \vec{o}_\psi, \vec{p}_\psi) \tag{4.15}$$

where:

- $\varphi_\psi \in \Phi_{\mathfrak{H}}$ *is a reference to the module type in the hardware model $\mathfrak{H}$.*
- $\hat{u}_\psi \in \mathcal{U}_{\mathfrak{H}}$ *is a reference to the ECU instance this module instance is attached to.*
- $\Theta_\psi \subseteq \mathcal{X}_{\mathfrak{P}}$ *is the set of* operating position instances *with $|\Theta_\psi| = |\Pi_{\varphi_\psi}|$. If multiple instances of the same module exist, it must be guaranteed that each operating position instance of every module instance is uniquely identified. All operating position instances are contained in an* object type *which is a partition of the set of* objects *of the* hardware model *of the plant. This means that operating position instances may be used as predicate parameters (unlike operating positions, which are not contained in $\mathcal{X}_{\mathfrak{P}}$).*
- $\vec{\theta}_\psi \colon \Pi_{\varphi_\psi} \to \Theta_\psi$ *is the bijective* operating position mapping function *that provides a one-to-one mapping between the operating positions of the respective module type φ_ψ and the operating position instances of the module instance ψ.*
- $\mathcal{C}_{\mathcal{I}\psi}$ *is the set of* initial module conditions*. These conditions are used to define the initial state of a module instance by setting the respective predicates (⊨ R5.2). An initial module condition conforms to the concept of conditions specified in Definition D3.*
- $\vec{\iota}_\psi \colon \mathcal{I}_{\varphi_\psi} \to \mathcal{I}_{u_\psi} \cup \{\perp\}$ *is the* input signal assignment function *that maps each of the input signals of the respective module type φ_ψ to a specific input channel of the respective ECU u_ψ, where $\perp$ is a special value that indicates that the respective input channel of the module instance is not connected.*

- $\vec{o}_\psi : \mathcal{O}_{\varphi_\psi} \rightarrow \mathcal{O}_{u_\psi} \cup \{\perp\}$ *is the* output signal assignment function *that maps each of the output signals of the respective module type φ_ψ to a specific output channel of the respective ECU u_ψ, where $\perp$ is a special value that indicates that the respective output channel of the module instance is not connected.*
- $\vec{p}_\psi : \mathcal{P}_{\varphi_\psi} \rightarrow \Sigma^*$ *is the* configurable parameter assignment function *that maps each of the configurable parameters of the respective module type φ_ψ to a specific value ($\vDash$ R5.4). Σ^* is defined as specified in equation (4.14).*

Definition D13 (overlapping operating position) *An* overlapping operating position *ℓ in L is a pair*[3]

$$\ell = (\theta_s, \theta_r) \tag{4.16}$$

where:

- $\theta_s \in \Pi$ *is the* source operating position instance.
- $\theta_r \in \Pi$ *is the* replacement operating position instance *with which the source operating position instance is to be unified.*

Semantically, an overlapping operating position indicates that operating position instance θ_r represents the same physical location than operating position instance θ_s in the plant, i.e., if a work piece is placed on or moved to θ_s, it is at the same time at θ_r. θ_s and θ_r typically belong to different module instances.

4.3. Formal Description of Automation Tasks

The goal of this section is to define the requirements for a task model that captures relevant aspects of MAL automation tasks on basis of the plant model introduced in the previous section. The task model is capable of representing functional and extra-functional requirements for task planning and execution.

4.3.1. Requirements

A typical automation task moves work pieces through a number of processing stations where the work pieces are handled according to a production plan. The approach followed in this does not specify the individual steps that are required to reach the production goal (i.e., how a work piece is treated), but the initial state and final state that should be reached (i.e., what should be the outcome) as well as additional constraints.

Requirement R10 (initial state) *The task model must be capable of representing the start-up state of the plant.*

[3] Notice that three operating positions θ_1, θ_2 and θ_3 can be declared as overlapping by recursively applying the overlapping relationship (e.g., $\mathsf{L} := ((\theta_1, \theta_2), (\theta_2, \theta_3))$).

Requirement R11 (goal state) *The task model must be capable of representing the goal state of the plant.*

By default, the synthesized strategy invokes behavioral interfaces one after another. This execution scheme is called *sequential execution*. For enhancing the throughput of the plant, *parallel execution* of behavioral interfaces is also considered.

Requirement R12 (parallel execution) *The task model must capture whether multiple behavioral interfaces should be executed in parallel, and if yes, at which degree.*

In addition, the optimality of multiple solutions satisfying the same specification should be assessed. The cost η_{b} specified for each behavioral interface $\mathsf{b} \in \mathcal{B}$ is used for this purpose.

Requirement R13 (quantitative analysis) [1]*The task model must be capable of specifying how a metric for the optimality of a synthesized strategy is derived from the costs of the individual behavioral interfaces.* [2]*In addition, an upper bound for the total worst case cost should be specified in the task model such that results exceeding a certain threshold do not qualify as a solution.*

4.3.2. Task Model

Definition D14 (task model) *A task model $\mathfrak{T}$ is a tuple*

$$\mathfrak{T} = (\mathfrak{P}_{\mathfrak{T}}, \mathcal{C}_{\mathcal{I}\mathfrak{T}}, \sigma_{\mathfrak{T}}, g_{\mathfrak{T}}, \eta_{\max\mathfrak{T}}, \odot_{\mathfrak{T}}, \otimes_{\mathfrak{T}}, d_{\mathfrak{T}}) \tag{4.17}$$

where:

- $\mathfrak{P}_{\mathfrak{T}}$ *is the* plant model *for task model* $\mathfrak{T}$*.*
- $\mathcal{C}_{\mathcal{I}\mathfrak{T}}$ *is the set of* initial task conditions *(⊨ R10). In contrast to initial conditions defined at module type level, these conditions typically indicate the initial state of the plant and the initial location of work pieces. Hence, they correspond to the initial assumptions in the production plan. An initial task condition conforms to the concept of conditions specified in Definition D3. It typically references objects defined in the plant model $\mathcal{X}_{\mathfrak{P}_{\mathfrak{T}}}$ and the plant's hardware model $\mathcal{X}_{\mathfrak{H}_{\mathfrak{P}_{\mathfrak{T}}}}$.*
- $\sigma_{\mathfrak{T}} \in \{\sigma_{\mathrm{r}}, \sigma_{\mathrm{rc}}, \sigma_{\mathrm{rp}}, \sigma_{\mathrm{rcp}}\}$ *specifies the type of solver to use for synthesizing a strategy. Table 4.1 summarizes the available solver types and which of the parameters in the task model they use as input. Notice that parallel action execution is supported for some solver types (⊨ R12).*
- $g_{\mathfrak{T}}$ *is the* goal state specification *(⊨ R11). It specifies the conditions that should apply as a result of executing the synthesized production plan, expressed by a respective valuation of state predicates. The valuation typically references objects from $\mathcal{X}_{\mathfrak{P}_{\mathfrak{T}}}$ and $\mathcal{X}_{\mathfrak{H}_{\mathfrak{P}_{\mathfrak{T}}}}$. As indicated later in Section 5.4.4.2, it corresponds to the reachability condition of the respective game. For example, the final properties of work pieces and modules may be stated. Formally, the goal state specification is a Boolean term, where the variables of the term are conditions $c = v(\mathsf{a}_{v_0}, \mathsf{a}_{v_1}, \ldots, \mathsf{a}_{v_{a-1}}) \in \mathcal{C}$ according to Definition D3. Since every*

Table 4.1.: Available solver types and used input parameters from the task model.

Solver type		**Used input parameters**				
Reachability…	$\sigma_{\mathfrak{T}}$	$g_{\mathfrak{T}}$	$\odot_{\mathfrak{T}}$	$\otimes_{\mathfrak{T}}$	$\eta_{\max\mathfrak{T}}$	$d_{\mathfrak{T}}$
only	σ_{r}	yes	no	no	no (∞)	1
+ bounded cost	σ_{rc}	yes	yes	no	yes ($<\infty$)	1
+ parallelization	σ_{rp}	yes	no	no	no (∞)	>1
+ bounded cost + parallelization	σ_{rcp}	yes	yes	yes	yes ($<\infty$)	>1

Boolean term can be written in disjunctive normal form, a goal state specification can be written as follows, where "(¬)" means that terms may appear in negated form:

$$g_{\mathfrak{T}} = \bigvee_i \bigwedge_j (\neg) v_{ij}(\mathsf{a}_{v_{ij_0}}, \mathsf{a}_{v_{ij_1}}, \ldots, \mathsf{a}_{v_{ij_{a-1}}}) \tag{4.18}$$

- $\eta_{\max\mathfrak{T}}$ *is the* cost bound *that must not be exceeded by the worst case execution run of the synthesized strategy (⊨ R13.2). The semantics of this cost value is user-defined and must match the semantics of the costs* η_{b} *annotated in the behavioral interfaces* $\mathsf{b} \in \mathcal{B}$*. The interpretation of the cost bound depends on* $\odot_{\mathfrak{T}}$ *and* $\otimes_{\mathfrak{T}}$*.*
- $\odot_{\mathfrak{T}}$ *is the* sequential composition operator*. It is used to calculate the combined cost* $\eta_{\mathsf{b}_1\to\ldots\to\mathsf{b}_{\beta+1}}$ *of a sequential invocation of behavioral interfaces* $\mathsf{b}_1, \mathsf{b}_2, \ldots, \mathsf{b}_\beta, \mathsf{b}_{\beta+1}, \beta > 0$ *from the cost* $\eta_{\mathsf{b}_1\to\ldots\to\mathsf{b}_\beta}$ *of an existing sequential execution trace of behavioral interface invocations* $\mathsf{b}_0, \mathsf{b}_1, \ldots, \mathsf{b}_{\beta-1}$ *and the cost* $\eta_{\mathsf{b}_{\beta+1}}$ *of another behavioral interface invocation* $\mathsf{b}_{\beta+1}$ *(⊨ R13.1).* $\odot_{\mathfrak{T}}$ *should be commutative, i.e., yield the same result independent of the order of input parameters. Currently supported values for* $\odot_{\mathfrak{T}}$ *are:*

$$\odot_{\mathfrak{T}} := \textit{sum} \quad \Rightarrow \quad \eta_{\mathsf{b}_1\to\ldots\to\mathsf{b}_{\beta+1}} := \eta_{\mathsf{b}_1\to\ldots\to\mathsf{b}_\beta} + \eta_{\mathsf{b}_{\beta+1}} \tag{4.19}$$

$$\odot_{\mathfrak{T}} := \textit{max} \quad \Rightarrow \quad \eta_{\mathsf{b}_1\to\ldots\to\mathsf{b}_{\beta+1}} := \max(\eta_{\mathsf{b}_1\to\ldots\to\mathsf{b}_\beta}, \eta_{\mathsf{b}_{\beta+1}}) \tag{4.20}$$

- $\otimes_{\mathfrak{T}}$ *is the* parallel composition operator*. It is used to calculate the combined cost* $\eta_{\mathsf{b}1\|\ldots\|\mathsf{b}_{\beta+1}}$ *of a parallel invocation of behavioral interfaces* $\mathsf{b}_1, \mathsf{b}_2, \ldots, \mathsf{b}_\beta, \mathsf{b}_{\beta+1}, \beta > 0$ *from the cost* $\eta_{\mathsf{b}_1\|\ldots\|\mathsf{b}_\beta}$ *of an existing parallel invocation of behavioral interfaces* $\mathsf{b}_0, \mathsf{b}_1, \ldots, \mathsf{b}_{\beta-1}$ *and the cost* $\eta_{\mathsf{b}_{\beta+1}}$ *of invoking another behavioral interface* $\mathsf{b}_{\beta+1}$ *in parallel (⊨ R13.1).* $\otimes_{\mathfrak{T}}$ *should be commutative, i.e., yield the same result independent of the order of input parameters. Currently supported values for* $\otimes_{\mathfrak{T}}$ *are:*

$$\otimes_{\mathfrak{T}} := \textit{sum} \quad \Rightarrow \quad \eta_{\mathsf{b}_1\|\ldots\|\mathsf{b}_{\beta+1}} := \eta_{\mathsf{b}_1\|\ldots\|\mathsf{b}_\beta} + \eta_{\mathsf{b}_{\beta+1}} \tag{4.21}$$

$$\otimes_{\mathfrak{T}} := \textit{max} \quad \Rightarrow \quad \eta_{\mathsf{b}_1\|\ldots\|\mathsf{b}_{\beta+1}} := \max(\eta_{\mathsf{b}_1\|\ldots\|\mathsf{b}_\beta}, \eta_{\mathsf{b}_{\beta+1}}) \tag{4.22}$$

- $d_{\mathfrak{T}} \in \mathbb{N}^+$ *is the maximum* degree of parallelization *(⊨ R12). It specifies how many actions should be executed in parallel at max. Typical values are 1 (no parallelization) and low values like 2 or 3, because synthesis time significantly increases with a higher degree of parallelization. Refer to Section 7.5 for a respective evaluation.*

4.3.3. Process Model

The process model is a complete model of hardware, plant and task. It specifies a concrete automation task to be executed in a concrete plant made out of concrete hardware modules.

Definition D15 (process model) *The set* $\mathfrak{M} = (\mathfrak{H}, \mathfrak{P}, \mathfrak{T})$ *with hardware model* $\mathfrak{H} = \mathfrak{H}_{\mathfrak{P}}$ *according to Definition D4, plant model* $\mathfrak{P} = \mathfrak{P}_{\mathfrak{T}}$ *according to Definition D10 and task model* $\mathfrak{T}$ *according to Definition D14 is called the* process model.

This definition concludes the formal language for specification of MALs and respective tasks. In the next section, it is applied to the running example.

4.4. Discussion and Application to Running Example

4.4.1. Formal Description of Hardware Modules from the Running Example

Using the provided model, the individual modules of the running example from Section 3.4 can be describes as follows. Define a new hardware model $\mathfrak{H}_0 = (\Phi_{\mathfrak{H}_0}, \mathcal{V}_{\mathfrak{H}_0}, \mathcal{X}_{\mathfrak{H}_0}, \mathcal{U}_{\mathfrak{H}_0})$ with initially empty sets for the elements of the tuple, i.e., $\Phi_{\mathfrak{H}_0} = \mathcal{V}_{\mathfrak{H}_0} = \mathcal{X}_{\mathfrak{H}_0} = \mathcal{U}_{\mathfrak{H}_0} = \emptyset$. In the following, those sets are incrementally extended as the hardware module types in the plant are modeled. Figure 4.1 depicts generalized representations of the respective hardware modules according to the model[4].

In order to improve readability, the set $\Pi \subseteq \mathcal{X}_{\mathfrak{H}_0}$ is used to refer to all operating positions in hardware model $\mathfrak{H}_0$. When items are added to Π, it is implicitly assumed that they are contained in $\mathcal{X}_{\mathfrak{H}_0}$, too. Furthermore, the set $\mathcal{W} \subset \mathcal{X}$ is used to refer to all work pieces in the plant. The content of this set is defined later in the plant model in Section 4.4.2 (see for example equation (4.96)).

4.4.1.1. Stacking Magazine

A stacking magazine (compare Figure 4.1 (a)) is a passive hardware module type with one operating position π_{bottom}. According to the assumptions in Section 3.4, a work piece is always available at position π_{bottom} when one is needed. Hence:

$$\Pi := \Pi \cup \{\pi_{\mathsf{bottom}}\} \tag{4.23}$$

$$\varphi_{\mathsf{Magazine}} := (\emptyset, \{\pi_{\mathsf{bottom}}\}, \emptyset, \emptyset, \emptyset) \tag{4.24}$$

$$\Phi_{\mathfrak{H}_0} := \Phi_{\mathfrak{H}_0} \cup \{\varphi_{\mathsf{Magazine}}\} \tag{4.25}$$

Equation (4.23) extends the (initially empty) set of operating positions by π_{bottom}. Equation (4.24) defines the stacking magazine module type with empty sets of behavioral

[4]These hardware modules correspond to the following *Festo Modular Production System (MPS)* [Fes12] components (compare Sections 7.5 and 7.6): 162 385 (stack magazine), 526 873 (branch module), 654 972 (rotary index table), 195 773 (testing module), 196 974 (drilling module), 532 934 (slide module).

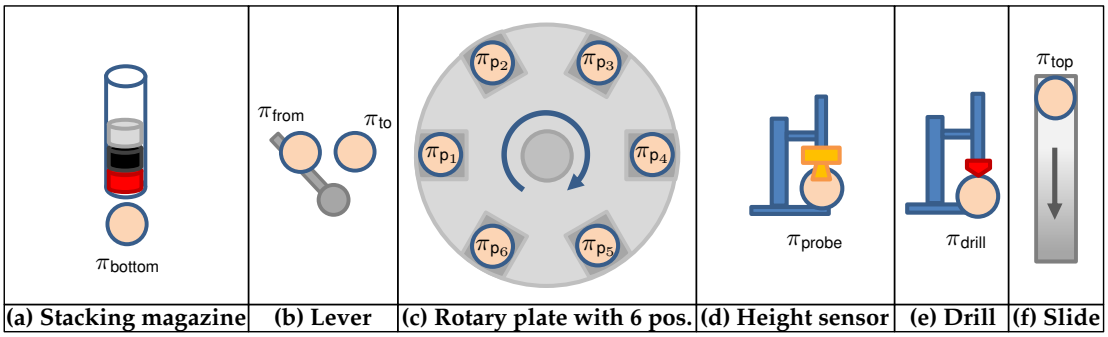


Figure 4.1.: Generalized representation of components from the running example.

interfaces, I/O signals and parameters (according to the passive nature of the module type, compare Definition D5 on page 48). Equation (4.25) extends the (initially empty) set of available module types in hardware model $\mathfrak{H}_0$ by $\varphi_{\text{Magazine}}$.

4.4.1.2. Lever

A lever (compare Figure 4.1 (b)) is an active hardware module with two operating positions π_{from} and π_{to}. For transferring a work piece between these positions, a behavioral interface $\mathsf{b}_{\text{lever-push}}$ is defined. In order to implement the behavioral interface, the position of work pieces needs to be formally represented. For this purpose two predicates are introduced:

$$\mathsf{at}\colon \mathcal{W} \times \Pi \to \mathbb{B} \tag{4.26}$$

$$\mathsf{occupied}\colon \Pi \to \mathbb{B} \tag{4.27}$$

$$\mathcal{V}_{\mathfrak{H}_0} := \mathcal{V}_{\mathfrak{H}_0} \cup \{\mathsf{at}, \mathsf{occupied}\} \tag{4.28}$$

The predicate $\mathsf{at}(w, \pi)$ denotes whether a given work piece $w \in \mathcal{W}$ is present at the position denoted by $\pi \in \Pi$.As mentioned before, $\mathcal{W}$ is defined later in the plant model. The predicate $\mathsf{occupied}(\pi)$ denotes whether a given operating position is occupied by a work piece. In this example, it is assumed that the following invariants always hold (compare Section 5.2):

$$\forall \pi \in \Pi\colon \mathsf{occupied}(\pi) \Leftrightarrow \exists w \in \mathcal{W}\colon \mathsf{at}(w, \pi) \tag{4.29}$$

$$\forall w_1, w_2 \in \mathcal{W}, \pi \in \Pi\colon \mathsf{at}(w_1, \pi) \wedge \mathsf{at}(w_2, \pi) \Leftrightarrow w_1 = w_2 \tag{4.30}$$

Equation (4.29) models that whenever an operating position is occupied, there exists a work piece that actually takes up this position and vice versa. Equation (4.30) specifies that each operating position can only be taken up by at most one work piece.

In addition to these predicates, one more predicate is required to represent where a lever is located in the assembly line. The reason for this is that when synthesizing a suitable strategy, we need to ensure that the solver may only pick the respective behavioral interface of the lever when there actually exists a lever in the plant between two

given operating positions. Hence, the lever-located predicate is introduced as follows:

$$\text{lever-located}\colon \Pi \times \Pi \to \mathbb{B} \tag{4.31}$$

$$\mathcal{V}_{\mathfrak{H}_0} := \mathcal{V}_{\mathfrak{H}_0} \cup \{\text{lever-located}\} \tag{4.32}$$

Recall that we did not introduce such a predicate for the stacking magazine. The reason for this is that the stacking magazine is a *passive* hardware module. Since such modules do not have any behavioral interfaces and hence the solver cannot choose a wrong behavioral interface for a given operating position, we can skip the addition of this predicate.

Based on the predicates defined above, the behavioral interface $\mathsf{b}_{\text{lever-push}}$ of the lever is defined. In this example, cost 1 is assumed for this behavioral interface.

$$\Pi := \Pi \cup \{\pi_{\text{from}}, \pi_{\text{to}}\} \tag{4.33}$$

$$\begin{aligned}\mathsf{b}_{\text{lever-push}} := ((&\mathsf{a}_{\text{obj}} \in \mathcal{W}, \mathsf{a}_{\text{from}} \in \Pi, \mathsf{a}_{\text{to}} \in \Pi), \mathcal{C}_{\mathcal{P}\mathsf{b}_{\text{lever-push}}}, \\ &\mathcal{E}_{\mathsf{b}_{\text{lever-push}}}, \emptyset, \emptyset, \hat{p}_{\text{lever-push}}, 1, 1)\end{aligned} \tag{4.34}$$

$$\mathcal{C}_{\mathcal{P}\mathsf{b}_{\text{lever-push}}} := \{\text{lever-located}(\mathsf{a}_{\text{from}}, \mathsf{a}_{\text{to}}), \text{at}(\mathsf{a}_{\text{obj}}, \mathsf{a}_{\text{from}}), \neg\text{occupied}(\mathsf{a}_{\text{to}})\} \tag{4.35}$$

$$\mathcal{E}_{\mathsf{b}_{\text{lever-push}}} := \{\neg\text{at}(\mathsf{a}_{\text{obj}}, \mathsf{a}_{\text{from}}), \text{at}(\mathsf{a}_{\text{obj}}, \mathsf{a}_{\text{to}}), \neg\text{occupied}(\mathsf{a}_{\text{from}}), \text{occupied}(\mathsf{a}_{\text{to}})\} \tag{4.36}$$

$$\varphi_{\text{Lever}} := (\{\mathsf{b}_{\text{lever-push}}\}, \{\pi_{\text{from}}, \pi_{\text{to}}\}, \emptyset, \{o_{\text{setLever}}\}, \{p_{\text{holdTime}}\}) \tag{4.37}$$

$$\Phi_{\mathfrak{H}_0} := \Phi_{\mathfrak{H}_0} \cup \{\varphi_{\text{Lever}}\} \tag{4.38}$$

Equation (4.33) extends the set of operating positions by π_{from} and π_{to}. Equation (4.34) defines the behavioral interface $\mathsf{b}_{\text{lever-push}}$ (compare Definition D6 on page 49) with the three arguments a_{obj}, a_{from} and a_{to} for moving the given object from π_{from} to π_{to}. The *primitive control function (PCF)* $\hat{p}_{\text{lever-push}}$ is triggered on the target platform to invoke the lever. The preconditions for the behavioral interface as defined in equation (4.35) are that a lever is located between π_{from} and π_{to}, that a work piece is located at π_{from} and that π_{to} is not occupied. The notation $\text{predicate}(\dots)$ is used to indicate that the respective predicate *has to be true* and $\neg\text{predicate}(\dots)$ to indicate that it *has to be false* in order to enable the behavioral interface. The effects of the behavioral interface specified in equation (4.36) are that the object is now located at π_{to} instead of π_{from}. The occupancy state is updated accordingly. The notation $\text{predicate}(\dots)$ is used to indicate that the respective predicate *is set to true* at the given parameters and $\neg\text{predicate}(\dots)$ to indicate that it *is set to false*. Equation (4.37) defines one output channel and one configurable parameter for the module type.

The digital output channel o_{setLever} is used to trigger the lever. The lever stays in the pushed position until the output signal is reset. The PCF $\hat{p}_{\text{lever-push}}$ that executes this behavioral interface implements the following functionality:

1. Set digital output o_{setLever} to high.
2. Wait p_{holdTime} milliseconds (interpreting the value of the parameter as a number).
3. Set digital output o_{setLever} to low.

4.4.1.3. Rotary Plate

A rotary plate with six operating positions (compare Figure 4.1 (c) on page 57) is an active hardware module with operating positions π_{p_1} through π_{p_6}. For reasons of simplicity, assume that only one work piece is located on any of the six operating positions at any point in time:

$$\forall i \in \{1, 2, \ldots, 6\} \colon \mathsf{occupied}(\pi_{\mathsf{p}_i}) \Rightarrow \forall j \in \{1, 2, \ldots, 6\} \setminus i : \neg\mathsf{occupied}(\pi_{\mathsf{p}_j}) \tag{4.39}$$

A more complicated model of this module type is required otherwise.

For rotating work pieces in clockwise direction, a behavioral interface $\mathsf{b}_{\mathsf{plate\text{-}rotate}}$ is defined. In order to implement the behavioral interface, we need to formally represent the relative position of operating positions on the plate. For this purpose, an additional predicate is introduced that models the fact that one operating position on the plate is followed by another operating position:

$$\mathsf{clockwise\text{-}next} \colon \Pi \times \Pi \to \mathbb{B} \tag{4.40}$$

$$\mathcal{V}_{\mathfrak{H}_0} := \mathcal{V}_{\mathfrak{H}_0} \cup \{\mathsf{clockwise\text{-}next}\} \tag{4.41}$$

Based on this predicate, the behavioral interface $\mathsf{b}_{\mathsf{plate\text{-}rotate}}$ of the rotary plate is defined. In this example, cost 1 is assumed for this behavioral interface.

$$\Pi := \Pi \cup \{\pi_{\mathsf{p}_1}, \ldots, \pi_{\mathsf{p}_6}\} \tag{4.42}$$

$$\begin{aligned} \mathsf{b}_{\mathsf{plate\text{-}rotate}} := ((\mathsf{a}_{\mathsf{obj}} \in \mathcal{W}, \mathsf{a}_{\mathsf{from}} \in \Pi, \mathsf{a}_{\mathsf{to}} \in \Pi), \mathcal{C}_{\mathcal{P}\mathsf{b}_{\mathsf{plate\text{-}rotate}}}, \\ \mathcal{E}_{\mathsf{b}_{\mathsf{plate\text{-}rotate}}}, \emptyset, \emptyset, \hat{p}_{\mathsf{plate\text{-}rotate}}, 1, 0) \end{aligned} \tag{4.43}$$

$$\mathcal{C}_{\mathcal{P}\mathsf{b}_{\mathsf{plate\text{-}rotate}}} := \{\mathsf{clockwise\text{-}next}(\mathsf{a}_{\mathsf{from}}, \mathsf{a}_{\mathsf{to}}), \mathsf{at}(\mathsf{a}_{\mathsf{obj}}, \mathsf{a}_{\mathsf{from}})\} \tag{4.44}$$

$$\mathcal{E}_{\mathsf{b}_{\mathsf{plate\text{-}rotate}}} := \{\neg\mathsf{at}(\mathsf{a}_{\mathsf{obj}}, \mathsf{a}_{\mathsf{from}}), \mathsf{at}(\mathsf{a}_{\mathsf{obj}}, \mathsf{a}_{\mathsf{to}}), \neg\mathsf{occupied}(\mathsf{a}_{\mathsf{from}}), \mathsf{occupied}(\mathsf{a}_{\mathsf{to}})\} \tag{4.45}$$

$$\begin{aligned} \varphi_{\mathsf{RotaryPlate}} := (\{\mathsf{b}_{\mathsf{plate\text{-}rotate}}\}, \{\pi_{\mathsf{p}_1}, \ldots, \pi_{\mathsf{p}_6}\}, \\ \{\iota_{\mathsf{isRotaryPlateInPosition}}\}, \{o_{\mathsf{setStartRotaryPlate}}\}, \emptyset) \end{aligned} \tag{4.46}$$

$$\Phi_{\mathfrak{H}_0} := \Phi_{\mathfrak{H}_0} \cup \{\varphi_{\mathsf{RotaryPlate}}\} \tag{4.47}$$

Due to the fact that, when triggering $\mathsf{b}_{\mathsf{plate\text{-}rotate}}$, the synthesis engine is free to select one of the six pairs of positions for $\mathsf{a}_{\mathsf{from}}$ and a_{to} that respect the clockwise-next predicate, a single pair of operating positions as a precondition in equation (4.44) is sufficient.

Notice the zero used as last element in $\mathsf{b}_{\mathsf{plate\text{-}rotate}}$ in equation (4.43), which indicates that parallelization of this behavioral interface is disabled. Hence, even if parallel execution of actions is enabled, this action will execute on its own. This restriction is set up to optimize the synthesis time at the cost of optimality of the resulting control program.

For simplicity, assume that the rotary plate is activated by the digital output channel $o_{\mathsf{setStartRotaryPlate}}$ that needs to be set to high in order to trigger a rotation by 60° in clockwise direction. The rotary plate automatically stops when it has reached the respective next position if the $o_{\mathsf{setStartRotaryPlate}}$ channel has been reset in the meanwhile. Further assume that the digital input channel $\iota_{\mathsf{isRotaryPlateInPosition}}$ is present that indicates when

the rotary plate has reached the respective next position[5]. The PCF $\hat{p}_{\text{plate-rotate}}$ that executes this behavioral interface implements the following functionality:

1. Set digital output $o_{\text{setStartRotaryPlate}}$ to high.
2. Wait[6] for digital input $\iota_{\text{isRotaryPlateInPosition}}$ to become low.
3. Set digital output $o_{\text{setStartRotaryPlate}}$ to low.
4. Wait for digital input $\iota_{\text{isRotaryPlateInPosition}}$ to become high.

4.4.1.4. Height Sensor Module

A height sensor (compare Figure 4.1 (d) on page 57) is an active hardware module with one operating position π_{probe}. A behavioral interface $\mathsf{b}_{\text{probe-height}}$ is defined for probing the height of a work piece at that position. In order to implement this behavioral interface, the height of work pieces needs to be formally represented. Without loss of generality, this example only distinguishes between two types of heights. For this purpose, introduce a new predicate:

$$\mathcal{X}_{\text{height}} := \{x_{\text{small}}, x_{\text{large}}\} \tag{4.48}$$

$$\text{height}\colon \mathcal{W} \times \mathcal{X}_{\text{height}} \to \mathbb{B} \tag{4.49}$$

$$\mathcal{X}_{\mathfrak{H}_0} := \mathcal{X}_{\mathfrak{H}_0} \cup \mathcal{X}_{\text{height}} \tag{4.50}$$

$$\mathcal{V}_{\mathfrak{H}_0} := \mathcal{V}_{\mathfrak{H}_0} \cup \{\text{height}\} \tag{4.51}$$

The predicate $\text{height}(w, h)$ evaluates to a true Boolean value if work piece w is of the height specified by h and false otherwise. This representation is favored over a simple predicate such as $\text{height-small}(w)$, because the height of a work piece may be initially unknown, which could not be represented with a predicate such as height-small.

Similar to the lever module, one more predicate is needed to represent where a height sensor module instance is located in the assembly line. Hence, introduce the height-sensor-located predicate as follows:

$$\text{height-sensor-located}\colon \Pi \to \mathbb{B} \tag{4.52}$$

$$\mathcal{V}_{\mathfrak{H}_0} := \mathcal{V}_{\mathfrak{H}_0} \cup \{\text{height-sensor-located}\} \tag{4.53}$$

Based on these predicates, the *sensor triggering* action $\mathsf{b}_{\text{probe-height}}$ of the height sensor module is defined according to Definition D8 on page 51. Cost for this behavioral interface is set to zero according to the constraints for sensor triggerings.

$$\Pi := \Pi \cup \{\pi_{\text{probe}}\} \tag{4.54}$$

$$\mathsf{b}_{\text{probe-height}} := ((\mathsf{a}_{\text{obj}} \in \mathcal{W}, \mathsf{a}_{\text{pos}} \in \Pi), \mathcal{C}_{\mathcal{P}\mathsf{b}_{\text{probe-height}}}, \emptyset, \emptyset, \mathcal{R}_{\mathsf{b}_{\text{probe-height}}}, \hat{p}_{\text{probe-height}}, 0, 1) \tag{4.55}$$

$$\mathcal{C}_{\mathcal{P}\mathsf{b}_{\text{probe-height}}} := \{\text{height-sensor-located}(\mathsf{a}_{\text{pos}}), \text{at}(\mathsf{a}_{\text{obj}}, \mathsf{a}_{\text{pos}})\} \tag{4.56}$$

$$\mathcal{R}_{\mathsf{b}_{\text{probe-height}}} := \{(0, \text{height}(\mathsf{a}_{\text{obj}}, x_{\text{large}})), (1, \text{height}(\mathsf{a}_{\text{obj}}, x_{\text{small}}))\} \tag{4.57}$$

[5]This design matches the design of the Festo MPS station 648 813 (processing), see also Section 7.3.
[6]No *busy* waiting is used in order to not constrain parallel execution of multiple behavioral interfaces.

$$\varphi_{\mathsf{HeightSensor}} := (\{\mathsf{b}_{\mathsf{probe\text{-}height}}\}, \{\pi_{\mathsf{probe}}\}, \{\iota_{\mathsf{isRodDown}}\}, \{o_{\mathsf{setRod}}\}, \{p_{\mathsf{probeTime}}\}) \tag{4.58}$$

$$\Phi_{\mathfrak{H}_0} := \Phi_{\mathfrak{H}_0} \cup \{\varphi_{\mathsf{HeightSensor}}\} \tag{4.59}$$

As long as the digital output channel o_{setRod} is high, an electromagnet pushes the probe rod down. If the probe rod is not blocked by the work piece (because it is small), then the digital input $\iota_{\mathsf{isRodDown}}$ becomes high. Otherwise, the $\iota_{\mathsf{isRodDown}}$ input stays low, which is also the default if o_{setRod} is low[7]. The PCF $\hat{p}_{\mathsf{probe\text{-}height}}$ that executes this behavioral interface implements the following functionality:

1. Set digital output o_{setRod} to high.
2. Wait p_{holdTime} milliseconds (interpreting the value of the parameter as a number).
3. Sample the value of digital input $\iota_{\mathsf{isRodDown}}$.
4. Set digital output o_{setRod} to low.
5. If the sampled $\iota_{\mathsf{isRodDown}}$ channel was high, return sensor result value 1, otherwise return sensor result value 0 (compare sensor result conditions in equation (4.57)).

4.4.1.5. Drill

A drill (compare Figure 4.1 (e) on page 57) is an active hardware module with one operating position π_{drill}. A behavioral interface $\mathsf{b}_{\mathsf{drill}}$ is defined for drilling a work piece at that position. For representing the effects of the drill, we introduce a new predicate drilled:

$$\mathsf{drilled}\colon \mathcal{W} \to \mathbb{B} \tag{4.60}$$

$$\mathcal{V}_{\mathfrak{H}_0} := \mathcal{V}_{\mathfrak{H}_0} \cup \{\mathsf{drilled}\} \tag{4.61}$$

The predicate $\mathsf{drilled}(w)$ evaluates to a true Boolean value if work piece w is drilled and false otherwise. Notice that unlike the height predicate introduced in equation (4.49), this representation requires the initial drilling state of a work piece to be known. Since all predicates are considered to be false for all valuations by default, this means that we consider all work piece to be initially undrilled[8].

Similar to the lever and height sensor modules, one more predicate is needed to represent where a drill is located in the assembly line. Hence, introduce the drill-located predicate as follows:

$$\mathsf{drill\text{-}located}\colon \Pi \to \mathbb{B} \tag{4.62}$$

$$\mathcal{V}_{\mathfrak{H}_0} := \mathcal{V}_{\mathfrak{H}_0} \cup \{\mathsf{drill\text{-}located}\} \tag{4.63}$$

Based on this predicate, formally define the behavioral interface $\mathsf{b}_{\mathsf{drill}}$ of the drill module. In this example, cost 1 is assumed for this behavioral interface.

[7] This design matches the design of the Festo MPS station 648 813 (processing), see also Section 7.3.

[8] If this semantics is not desired, the same approach used in the height predicate can be followed by defining a predicate $\mathsf{drilled\text{-}state}(w, x)$ with $x \in \{x_{\mathsf{drilled}}, x_{\mathsf{undrilled}}\}$ and $\mathcal{X}_{\mathfrak{H}_0} := \mathcal{X}_{\mathfrak{H}_0} \cup \{x_{\mathsf{drilled}}, x_{\mathsf{undrilled}}\}$.

$$\Pi := \Pi \cup \{\pi_{\mathsf{drill}}\} \tag{4.64}$$
$$\mathsf{b}_{\mathsf{drill}} := ((\mathsf{a}_{\mathsf{obj}} \in \mathcal{W}, \mathsf{a}_{\mathsf{pos}} \in \Pi), \mathcal{C}_{\mathcal{P}\,\mathsf{b}_{\mathsf{drill}}}, \mathcal{E}_{\mathsf{b}_{\mathsf{drill}}}, \emptyset, \emptyset, \hat{p}_{\mathsf{drill}}, 1, 1) \tag{4.65}$$
$$\mathcal{C}_{\mathcal{P}\,\mathsf{b}_{\mathsf{drill}}} := \{\mathsf{drill\text{-}located}(\mathsf{a}_{\mathsf{pos}}), \mathsf{at}(\mathsf{a}_{\mathsf{obj}}, \mathsf{a}_{\mathsf{pos}})\} \tag{4.66}$$
$$\mathcal{E}_{\mathsf{b}_{\mathsf{drill}}} := \{\mathsf{drilled}(\mathsf{a}_{\mathsf{obj}})\} \tag{4.67}$$
$$\varphi_{\mathsf{Drill}} := (\{\mathsf{b}_{\mathsf{drill}}\}, \{\pi_{\mathsf{drill}}\}, \{\iota_{\mathsf{isDrillDown}}, \iota_{\mathsf{isDrillUp}}\}, \{o_{\mathsf{setClamp}}, o_{\mathsf{setDrill}}, o_{\mathsf{setMoveDrillDown}}, o_{\mathsf{setMoveDrillUp}}\}, \{p_{\mathsf{drillTime}}\}) \tag{4.68}$$
$$\Phi_{\mathfrak{H}_0} := \Phi_{\mathfrak{H}_0} \cup \{\varphi_{\mathsf{Drill}}\} \tag{4.69}$$

The following digital I/O channels are involved: the $o_{\mathsf{setMoveDrillDown}}$ and $o_{\mathsf{setMoveDrillUp}}$ outputs are used to make the linear axis on which the drill is mounted to go down and up, respectively. Only one of them may be high at any one time. If both outputs are low, the linear axis does not move. The $\iota_{\mathsf{isDrillDown}}$ and $\iota_{\mathsf{isDrillUp}}$ inputs indicate that the respective position is reached. The o_{setClamp} output is used to activate a clamp that fixes the work piece in place for drilling. Finally, the o_{setDrill} output controls whether the drill is on or off[9]. The PCF $\hat{p}_{\mathsf{drill}}$ that executes this behavioral interface implements the following functionality (we implicitly assume that the linear axis of the drill is initially up):

1. Set digital output o_{setClamp} to high.
2. Set digital output o_{setDrill} to high.
3. Set digital output $o_{\mathsf{setMoveDrillUp}}$ to low and $o_{\mathsf{setMoveDrillDown}}$ to high.
4. Wait for digital input $\iota_{\mathsf{isDrillDown}}$ to become high.
5. Wait $p_{\mathsf{drillTime}}$ milliseconds (interpreting the value of the parameter as a number).
6. Set digital output $o_{\mathsf{setMoveDrillDown}}$ to low and $o_{\mathsf{setMoveDrillUp}}$ to high.
7. Wait for digital input $\iota_{\mathsf{isDrillUp}}$ to become high.
8. Set digital output o_{setDrill} to low.
9. Set digital output o_{setClamp} to low.

4.4.1.6. Slide

A slide (compare Figure 4.1 (f) on page 57) is a passive hardware module with one operating position π_{top}. According to the description in Section 3.4, we can assume that a work piece dropped at π_{top} will vanish immediately. Hence:

$$\Pi := \Pi \cup \{\pi_{\mathsf{top}}\} \tag{4.70}$$
$$\varphi_{\mathsf{Slide}} = (\emptyset, \{\pi_{\mathsf{top}}\}, \emptyset, \emptyset, \emptyset) \tag{4.71}$$
$$\Phi_{\mathfrak{H}_0} := \Phi_{\mathfrak{H}_0} \cup \{\varphi_{\mathsf{Slide}}\} \tag{4.72}$$

[9] This design matches the design of the Festo MPS station 648 813 (processing), see also Section 7.3.

4.4.1.7. ECU Types

Finally, the supported ECU types are formally represented. In our example, two ECU of type *Siemens S7-313C* are used, which have the following properties:

- 16 digital input channels organized in two bytes with 8 channels each.
- 16 digital output channels organized in two bytes with 8 channels each.
- Ability to be "remote controlled" via Siemens *Multi-Point Interface (MPI)* and *open platform communications (OPC)* [IL01].

Hence, the formal model of the ECU type is defines as follows:

$$\begin{aligned} u_{\text{S7-313C}} := (&\text{Siemens-S7}, \{\iota_{0.0}, \iota_{0.1}, \ldots, \iota_{0.7}, \iota_{1.0}, \iota_{1.1}, \ldots, \iota_{1.7}\}, \\ &\{o_{0.0}, o_{0.1}, \ldots, o_{0.7}, o_{1.0}, o_{1.1}, \ldots, o_{1.7}\}, \\ &\{p_{\text{mpiAddress}}\}) \end{aligned} \tag{4.73}$$

$$\mathcal{U}_{\mathfrak{H}_0} := \mathcal{U}_{\mathfrak{H}_0} \cup \{u_{\text{S7-313C}}\} \tag{4.74}$$

Equation (4.73) specifies that the ECU conforms to the platform type Siemens-S7, which is one of the supported target systems. The configurable ECU parameter $p_{\text{mpiAddress}}$ represents the address of the ECU on the MPI bus which it is attached to.

4.4.1.8. Summary

Here is a summary of *hardware model* $\mathfrak{H}_0$:

$$\Phi_{\mathfrak{H}_0} = \{\varphi_{\text{Magazine}}, \varphi_{\text{Lever}}, \varphi_{\text{RotaryPlate}}, \varphi_{\text{HeightSensor}}, \varphi_{\text{Drill}}, \varphi_{\text{Slide}}\} \tag{4.75}$$

$$\begin{aligned} \mathcal{V}_{\mathfrak{H}_0} = \{&\text{at}, \text{occupied}, \text{lever-located}, \text{clockwise-next}, \\ &\text{height}, \text{height-sensor-located}, \text{drilled}, \text{drill-located}\} \end{aligned} \tag{4.76}$$

$$\mathcal{X}_{\mathfrak{H}_0} = \{\pi_{\text{bottom}}, \pi_{\text{from}}, \pi_{\text{to}}, \pi_{\text{p}_1}, \ldots, \pi_{\text{p}_6}, \pi_{\text{probe}}, \pi_{\text{drill}}, \pi_{\text{top}}, x_{\text{small}}, x_{\text{large}}\} \tag{4.77}$$

$$\mathcal{U}_{\mathfrak{H}_0} = \{u_{\text{S7-313C}}\} \tag{4.78}$$

$$\bigcup_{\mathcal{B}_\varphi, \varphi \in \Phi_{\mathfrak{H}_0}} = \{\text{b}_{\text{lever-push}}, \text{b}_{\text{plate-rotate}}, \text{b}_{\text{probe-height}}, \text{b}_{\text{drill}}\} \tag{4.79}$$

4.4.2. Formal Description of the Assembly Line from the Running Example

Using hardware model $\mathfrak{H}_0$, the concrete assembly line from the running example in Section 3.4 is described in a plant model $\mathfrak{P}_0$ as follows:

$$\mathfrak{P}_0 = (\mathfrak{H}_0, \mathcal{X}_{\mathfrak{P}_0}, \Psi_{\mathfrak{P}_0}, \widehat{\mathcal{U}}_{\mathfrak{P}_0}, \mathsf{L}_{\mathfrak{P}_0}) \tag{4.80}$$

The individual elements of this tuple are defined in the following. First, consider the set of ECU instances $\widehat{\mathcal{U}}_{\mathfrak{P}_0}$. In the running example, two ECUs are used to control the module instances.

$$\widehat{\mathcal{U}}_{\mathfrak{P}_0} := \{\hat{u}_{\mathsf{Plant}}, \hat{u}_{\mathsf{Drill}}\} \tag{4.81}$$
$$\hat{u}_{\mathsf{Plant}} := (u_{\mathsf{S7\text{-}313C}}, \vec{u}_{\mathsf{Plant}}) \tag{4.82}$$
$$\hat{u}_{\mathsf{Drill}} := (u_{\mathsf{S7\text{-}313C}}, \vec{u}_{\mathsf{Drill}}) \tag{4.83}$$
$$\vec{u}_{\mathsf{Plant}} := \{p_{\mathsf{mpiAddress}} \mapsto 1\} \tag{4.84}$$
$$\vec{u}_{\mathsf{Drill}} := \{p_{\mathsf{mpiAddress}} \mapsto 2\} \tag{4.85}$$

The mapping of the configurable parameter $p_{\mathsf{mpiAddress}}$ corresponds to the example setup. Next, the set of operating position *instances* required to represent two levers and one instance of the other module types needs to be defined. The notation $\pi_{\mathsf{SM1.bottom}}$ is a short form to refer to the operating position instance $\psi_{\mathsf{SM1}}.\pi_{\mathsf{bottom}}$.

$$\begin{aligned}\mathcal{X}_{\mathfrak{P}_0} := \{&\pi_{\mathsf{SM1.bottom}}, \pi_{\mathsf{L1.from}}, \pi_{\mathsf{L1.to}}, \pi_{\mathsf{RP1.p}_1}, \ldots, \pi_{\mathsf{RP1.p}_6},\\ &\pi_{\mathsf{H1.probe}}, \pi_{\mathsf{D1.drill}}, \pi_{\mathsf{L2.from}}, \pi_{\mathsf{L2.to}}, \pi_{\mathsf{S1.top}}\}\end{aligned} \tag{4.86}$$

Subsequently, the respective module instances need to be defined:

$$\Psi_{\mathfrak{P}_0} := \{\psi_{\mathsf{SM1}}, \psi_{\mathsf{L1}}, \psi_{\mathsf{RP1}}, \psi_{\mathsf{H1}}, \psi_{\mathsf{D1}}, \psi_{\mathsf{L2}}, \psi_{\mathsf{S1}}\} \tag{4.87}$$

Recall that items from this set are defined according to Definition D12 on page 52:

$$\psi = (\varphi_\psi, u_\psi, \Theta_\psi, \vec{\theta}_\psi, \mathcal{C}_{\mathcal{I}\psi}, \vec{\iota}_\psi, \vec{o}_\psi, \vec{p}_\psi)$$

φ_ψ and u_ψ reference the module type and the ECU instance of the module instance. Θ_ψ is the set of *operating position instances* and $\vec{\theta}_\psi$ is the function that maps operating positions of the module type to the module instance. $\mathcal{C}_{\mathcal{I}\psi}$ is the set of *initial module conditions* for module instance ψ. $\vec{\iota}_\psi$, $\vec{o}_\psi$ and $\vec{p}_\psi$ are the functions that map I/O signals and parameters to the respective channels and values, respectively.

Before defining the respective module instances, it is worth to mention how the $\mathcal{C}_{\mathcal{I}\psi}$ argument works. In order for the solver to not parameterize a behavioral interface invocation with a wrong set of operating positions (e.g., attempting to trigger a lever between operating positions π_{bottom} and π_{top}), the initial module conditions initialize the predicates that were introduced to specify valid combinations of operating position arguments to behavioral interfaces. Those predicates have been defined in the equations (4.31), (4.40), (4.52) and (4.62). They are used as preconditions in the respective behavioral interfaces and hence constrain the solver to choose positional arguments only for valid combinations of operating positions. With this background knowledge, the definition of the module instances is straightforward[10]:

[10] We set $\vec{\iota}_\psi$, $\vec{o}_\psi$ and $\vec{p}_\psi$ to $\perp$ if the domain of the respective functions is empty.

Table 4.2.: Digital input channel assignment in the running example.

ECU	Byte	Digital input channels							
		0	1	2	3	4	5	6	7
$\hat{u}_{\mathsf{Plant}}$	ι_0	–	$\iota_{\mathsf{isRotaryPlateInPosition}}$	$\iota_{\mathsf{isRodDown}}$	–	–	–	–	–
	ι_1	–	–	–	–	–	–	–	–
$\hat{u}_{\mathsf{Drill}}$	ι_0	–	–	–	$\iota_{\mathsf{isDrillDown}}$	$\iota_{\mathsf{isDrillUp}}$	–	–	–
	ι_1	–	–	–	–	–	–	–	–

$$\psi_{\mathsf{SM1}} = (\varphi_{\mathsf{Magazine}}, \hat{u}_{\mathsf{Plant}}, \{\pi_{\mathsf{SM1.bottom}}\}, \{\pi_{\mathsf{bottom}} \mapsto \pi_{\mathsf{SM1.bottom}}\}, \emptyset, \bot, \bot, \bot) \tag{4.88}$$

$$\begin{aligned}\psi_{\mathsf{L1}} = (&\varphi_{\mathsf{Lever}}, \hat{u}_{\mathsf{Plant}}, \{\pi_{\mathsf{L1.from}}, \pi_{\mathsf{L1.to}}\}, \{\pi_{\mathsf{from}} \mapsto \pi_{\mathsf{L1.from}}, \pi_{\mathsf{to}} \mapsto \pi_{\mathsf{L1.to}}\},\\ &\{\mathsf{lever\text{-}located}(\pi_{\mathsf{L1.from}}, \pi_{\mathsf{L1.to}})\}, \bot, \{o_{\mathsf{setLever}} \mapsto o_{0.0}\},\\ &\{p_{\mathsf{holdTime}} \mapsto 1000\,\mathrm{ms}\})\end{aligned} \tag{4.89}$$

$$\begin{aligned}\psi_{\mathsf{RP1}} = (&\varphi_{\mathsf{RotaryPlate}}, \hat{u}_{\mathsf{Plant}}, \{\pi_{\mathsf{RP1.p}_1}, \ldots, \pi_{\mathsf{RP1.p}_6}\}, \{\pi_{\mathsf{p}_1} \mapsto \pi_{\mathsf{RP1.p}_1}, \ldots, \pi_{\mathsf{p}_6} \mapsto \pi_{\mathsf{RP1.p}_6}\},\\ &\{\mathsf{clockwise\text{-}next}(\pi_{\mathsf{RP1.p}_1}, \pi_{\mathsf{RP1.p}_2}), \mathsf{clockwise\text{-}next}(\pi_{\mathsf{RP1.p}_2}, \pi_{\mathsf{RP1.p}_3}),\\ &\ \mathsf{clockwise\text{-}next}(\pi_{\mathsf{RP1.p}_3}, \pi_{\mathsf{RP1.p}_4}), \mathsf{clockwise\text{-}next}(\pi_{\mathsf{RP1.p}_4}, \pi_{\mathsf{RP1.p}_5}),\\ &\ \mathsf{clockwise\text{-}next}(\pi_{\mathsf{RP1.p}_5}, \pi_{\mathsf{RP1.p}_6}), \mathsf{clockwise\text{-}next}(\pi_{\mathsf{RP1.p}_6}, \pi_{\mathsf{RP1.p}_1})\},\\ &\{\iota_{\mathsf{isRotaryPlateInPosition}} \mapsto \iota_{0.1}\}, \{o_{\mathsf{setStartRotaryPlate}} \mapsto o_{0.1}\}, \bot)\end{aligned} \tag{4.90}$$

$$\begin{aligned}\psi_{\mathsf{H1}} = (&\varphi_{\mathsf{HeightSensor}}, \hat{u}_{\mathsf{Plant}}, \{\pi_{\mathsf{H1.probe}}\}, \{\pi_{\mathsf{probe}} \mapsto \pi_{\mathsf{H1.probe}}\},\\ &\{\mathsf{height\text{-}sensor\text{-}located}(\pi_{\mathsf{H1.probe}})\}, \{\iota_{\mathsf{isRodDown}} \mapsto \iota_{0.2}\}, \{o_{\mathsf{setRod}} \mapsto o_{0.2}\},\\ &\{p_{\mathsf{holdTime}} \mapsto 200\,\mathrm{ms}\})\end{aligned} \tag{4.91}$$

$$\begin{aligned}\psi_{\mathsf{D1}} = (&\varphi_{\mathsf{Drill}}, \hat{u}_{\mathsf{Drill}}, \{\pi_{\mathsf{D1.drill}}\}, \{\pi_{\mathsf{drill}} \mapsto \pi_{\mathsf{D1.drill}}\}, \{\mathsf{drill\text{-}located}(\pi_{\mathsf{D1.drill}})\},\\ &\{\iota_{\mathsf{isDrillDown}} \mapsto \iota_{0.3}, \iota_{\mathsf{isDrillUp}} \mapsto \iota_{0.4}\}, \{o_{\mathsf{setClamp}} \mapsto o_{0.0}, o_{\mathsf{setDrill}} \mapsto o_{0.1},\\ &\ o_{\mathsf{setMoveDrillDown}} \mapsto o_{0.2}, o_{\mathsf{setMoveDrillUp}} \mapsto o_{0.3}\}, \{p_{\mathsf{drillTime}} \mapsto 1000\,\mathrm{ms}\})\end{aligned} \tag{4.92}$$

$$\begin{aligned}\psi_{\mathsf{L2}} = (&\varphi_{\mathsf{Lever}}, \hat{u}_{\mathsf{Plant}}, \{\pi_{\mathsf{L2.from}}, \pi_{\mathsf{L2.to}}\}, \{\pi_{\mathsf{from}} \mapsto \pi_{\mathsf{L2.from}}, \pi_{\mathsf{to}} \mapsto \pi_{\mathsf{L2.to}}\},\\ &\{\mathsf{lever\text{-}located}(\pi_{\mathsf{L2.from}}, \pi_{\mathsf{L2.to}})\}, \bot, \{o_{\mathsf{setLever}} \mapsto o_{0.3}\},\\ &\{p_{\mathsf{holdTime}} \mapsto 1000\,\mathrm{ms}\})\end{aligned} \tag{4.93}$$

$$\psi_{\mathsf{S1}} = (\varphi_{\mathsf{Slide}}, \hat{u}_{\mathsf{Plant}}, \{\pi_{\mathsf{S1.top}}\}, \{\pi_{\mathsf{top}} \mapsto \pi_{\mathsf{S1.top}}\}, \emptyset, \bot, \bot, \bot) \tag{4.94}$$

Tables 4.2 and 4.3 summarize the digital I/O channel assignment of the ECU in the running example, where "–" indicates that the respective channel is unused. Finally, the arrangement of module instances is specified according to Definition 13 on page 53:

$$\begin{aligned}\mathsf{L}_{\mathfrak{P}_0} := (&(\pi_{\mathsf{SM1.bottom}}, \pi_{\mathsf{L1.from}}), (\pi_{\mathsf{L1.to}}, \pi_{\mathsf{RP1.p}_1}), (\pi_{\mathsf{RP1.p}_2}, \pi_{\mathsf{H1.probe}}),\\ &(\pi_{\mathsf{RP1.p}_3}, \pi_{\mathsf{D1.drill}}), (\pi_{\mathsf{RP1.p}_4}, \pi_{\mathsf{L2.from}}), (\pi_{\mathsf{L2.to}}, \pi_{\mathsf{S1.top}}))\end{aligned} \tag{4.95}$$

Each pair of operating positions in this set represents the same physical location in the plant. Hence, the possible material flow between module instances is implicitly specified.

Table 4.3.: Digital output channel assignment in the running example.

ECU	Byte	Digital output channels				
		0	1	2	3	4 . . . 7
$\hat{u}_{\mathsf{Plant}}$	o_0	$o_{\mathsf{setLever\ (L1)}}$	$o_{\mathsf{setStartRotaryPlate}}$	o_{setRod}	$o_{\mathsf{setLever\ (L2)}}$	–
	o_1	–	–	–	–	–
$\hat{u}_{\mathsf{Drill}}$	o_0	o_{setClamp}	o_{setDrill}	$o_{\mathsf{setMoveDrillDown}}$	$o_{\mathsf{setMoveDrillUp}}$	–
	o_1	–	–	–	–	–

For describing automation tasks on work pieces, they need to be represented as objects in the model. Without loss of generality, define a set of work piece objects $\mathcal{W}$ with the objects x_{wp1} (work piece 1) and x_{wp2} (work piece 2) and add it to the set of objects in the plant model:

$$\mathcal{W} := \{x_{\mathsf{wp1}}, x_{\mathsf{wp2}}\} \tag{4.96}$$

$$\mathcal{X}_{\mathfrak{P}_0} := \mathcal{X}_{\mathfrak{P}_0} \cup \mathcal{W} \tag{4.97}$$

This allows to reason about the work pieces, for example in (goal) conditions like $\mathsf{drilled}(x_{\mathsf{wp1}})$ and $\mathsf{at}(x_{\mathsf{wp2}}, \pi_{\mathsf{top}})$.

4.4.3. Task Description for the Running Example

This section defines example task models for plant model $\mathfrak{P}_0$ that correspond to different automation tasks in the running example. The task models are referred to in subsequent sections of this work. Recall that a task specification consists of the following elements according to Definition D14 on page 54:

$$\mathfrak{T} = (\mathfrak{P}_{\mathfrak{T}}, \mathcal{C}_{\mathcal{I}\mathfrak{T}}, \sigma_{\mathfrak{T}}, g_{\mathfrak{T}}, \eta_{\max\mathfrak{T}}, \odot_{\mathfrak{T}}, \otimes_{\mathfrak{T}}, d_{\mathfrak{T}})$$

where $\mathfrak{P}_{\mathfrak{T}}$ is the corresponding plant model, $\mathcal{C}_{\mathcal{I}\mathfrak{T}}$ is the set of initial task conditions, $\sigma_{\mathfrak{T}}$ the solver type, $g_{\mathfrak{T}}$ the goal condition, $\eta_{\max\mathfrak{T}}$ the cost bound, $\odot_{\mathfrak{T}}$ and $\otimes_{\mathfrak{T}}$ the sequential and parallel composition operator and $d_{\mathfrak{T}}$ the maximum degree of parallelization. Notice that the following examples are just for illustration purposes, refer to Chapter 7 for more in-depth evaluations.

4.4.3.1. Task 1: Drill Work Pieces

Informal task description:

Drill a work piece x_{wp1} initially located at the stacking magazine and move it to the slide.

Formal task description:

$$\mathfrak{T}_1 = (\mathfrak{P}_0, \mathcal{C}_{\mathcal{I}1}, \sigma_1, g_1, \bot, \bot, \bot, d_1) \tag{4.98}$$
$$\mathcal{C}_{\mathcal{I}1} = \{\mathsf{at}(x_{\mathsf{wp1}}, \pi_{\mathsf{SM1.bottom}}), \mathsf{occupied}(\pi_{\mathsf{SM1.bottom}})\} \tag{4.99}$$
$$g_1 = \mathsf{at}(x_{\mathsf{wp1}}, \pi_{\mathsf{S1.top}}) \wedge \mathsf{drilled}(x_{\mathsf{wp1}}) \tag{4.100}$$

In this example, assume $\sigma_1 \in \{\sigma_\mathrm{r}, \sigma_\mathrm{rp}\}$ ("reachability only" or "reachability + parallelization"). Parameters $\eta_{\max 1}$, $\odot_1$ and $\otimes_1$ are set to $\bot$, because these solvers ignore them anyway (compare Table 4.1 on page 55). Notice that we do not explicitly represent the fact that a drilled work piece does not need to be drilled again. This is fine as long as excessive drilling does not cause any harm and we do not consider the optimality of the synthesized control program. This fact leads to the synthesized control program to be purely deterministic, because no sensors need to be triggered in order to achieve the task. Algorithm 4.1 shows a simplified version of a suitable centralized control program in pseudocode. In the algorithm, $\mathsf{action}(\dots)$ is used as a short-hand for the invocation of the primitive control function $\hat{p}_{\mathsf{b_{action}}}(\dots)$ for behavioral interface $\mathsf{b_{action}}$ with the given parameters.

4.4.3.2. Task 2: Drill Undrilled Work Pieces

In the next step, consider an example that requires sensor triggerings.

Informal task description:

Transport a work piece x_{wp1} initially located at the stacking magazine to the slide. If x_{wp1} is small, drill it.

Formal task description:

$$\mathfrak{T}_2 = (\mathfrak{P}_0, \mathcal{C}_{\mathcal{I}2}, \sigma_2, g_2, \bot, \bot, \bot, d_2) \tag{4.101}$$
$$\mathcal{C}_{\mathcal{I}2} = \{\mathsf{at}(x_{\mathsf{wp1}}, \pi_{\mathsf{SM1.bottom}}), \mathsf{occupied}(\pi_{\mathsf{SM1.bottom}})\} \tag{4.102}$$
$$g_2 = \mathsf{at}(x_{\mathsf{wp1}}, \pi_{\mathsf{S1.top}}) \wedge (\mathsf{drilled}(x_{\mathsf{wp1}}) \Leftrightarrow \mathsf{height}(x_{\mathsf{wp1}}, x_{\mathsf{small}})) \tag{4.103}$$

In this example, assume $\sigma_2 \in \{\sigma_\mathrm{r}, \sigma_\mathrm{rp}\}$. The difference between $\mathfrak{T}_1$ and $\mathfrak{T}_2$ is subtle: since the value of $\mathsf{height}(x_{\mathsf{wp1}}, x_{\mathsf{small}})$ is not known in advance, this specification requires sensory input from the height sensor module[11]. Hence, the suitable centralized control program shown in Algorithm 4.2 contains a guard for executing the drill operation. Notice that this approach assumes all predicate valuations that are not explicitly specified as initial conditions to be set to false. For example, the initial condition $\neg\mathsf{drilled}(x_{\mathsf{wp1}})$ implicitly holds. In order to match this expectation, no work pieces that are already drilled may be inserted into the stacking magazine, otherwise the initial assumptions in the model would not match the "real world".

[11]Notice that the solver being used in this work requires the additional term $\mathsf{height}(x_{\mathsf{wp1}}, x_{\mathsf{small}}) \vee$

4.4.4. Semantics of Cost in Quantitative Synthesis

In order to support quantitative evaluation of synthesized control programs, a notion of cost has been introduced for behavioral interfaces. The two operators sequential composition ($\odot$) and parallel composition ($\otimes$) were introduced to formulate how the total cost is calculated from the individual costs.

Tables 4.4 and 4.5 provide intuitive examples for semantics of cost with respect to the choice of the composition operators. Table 4.4 assumes that cost corresponds to *execution time (ET)* and Table 4.5 assumes that it corresponds to power consumption. Table cells marked with "–" indicate that no intuitive example was found. See Chapter 7 and [CGB13] for more details.

4.4.5. Limitations of Modeling

Although the presented modeling language covers a number of aspects, its expressiveness is not ideal for certain use cases which are listed in the following.

The *hardware model* should be as generic as possible to allow a control engineer to use the modular model elements in a large variety of concrete automation tasks. However, the more generic the modules and their behavioral interfaces in the hardware model are specified, the harder it is to connect the respective state variables with each other. In the running example, the generic predicates at and occupied were defined that are shared among all behavioral interfaces. These predicates are quite generic and it is meaningful to share them between the behavioral interfaces, but when it comes to less generic shared predicates, the question arises what is the right amount of interdependencies between behavioral interfaces. If behavioral interfaces are too loosely coupled, the number of predicates is high (increasing synthesis time) and the effects that one predicate valuation has on another one needs to be represented in what we call *spontaneous state changes*, i.e., behavioral interfaces with disjoint predicates in their preconditions and effects and an "empty" PCF. Although they feel like environment moves, spontaneous state changes should be modeled as Controller moves in order to not increase the number of sensor inputs, which can increase synthesis time.

A related issue is concerned with modeling the order of execution of behavioral interfaces. Consider an automation task where one processing step on a certain work piece needs to be guaranteed to be performed before another processing step. A simple example would be the mounting of a screw in a hole that first needs to be drilled. In this example, the predicate $\mathsf{drilled}(x_{\mathsf{wp1}})$ can be used as a precondition for the behavioral interface $\mathsf{b}_{\mathsf{mount\text{-}screw}}$, because it is not meaningful to mount a screw in an undrilled work piece. However, order of execution of behavioral interfaces is not always as easily defined. Consider the automation task where a work piece should be painted after a hole has been drilled. In this case, $\mathsf{drilled}(x_{\mathsf{wp1}})$ should not be a precondition for behavioral interface $\mathsf{b}_{\mathsf{paint}}$, because there may well be use cases of the same *hardware model*

$\mathsf{height}(x_{\mathsf{wp1}}, x_{\mathsf{large}})$ to be added to the goal condition as a conjunction in this case (otherwise it does not find a solution). This term is omitted here for brevity.

Algorithm 4.1: Suitable centralized control program for task model $\mathfrak{T}_1$.

```
1 lever-push (x_wp1, π_L1.from, π_L1.to);
2 plate-rotate (x_wp1, π_RP1.p1, π_RP1.p2);
3 plate-rotate (x_wp1, π_RP1.p2, π_RP1.p3);
4 drill (x_wp1, π_D1.drill);
5 plate-rotate (x_wp1, π_RP1.p3, π_RP1.p4);
6 lever-push (x_wp1, π_L2.from, π_L2.to);
```

Algorithm 4.2: Suitable centralized control program for task model $\mathfrak{T}_2$.

```
1 lever-push (x_wp1, π_L1.from, π_L1.to);
2 plate-rotate (x_wp1, π_RP1.p1, π_RP1.p2);
3 probe-height (x_wp1, π_H1.probe);
4 plate-rotate (x_wp1, π_RP1.p2, π_RP1.p3);
5 if height (x_wp1, x_small) then
6 |   drill (x_wp1, π_D1.drill);
7 end
8 plate-rotate (x_wp1, π_RP1.p3, π_RP1.p4);
9 lever-push (x_wp1, π_L2.from, π_L2.to);
```

Table 4.4.: Semantics of composition operators when cost equals execution time.

cost ≈ execution time	$\odot :=$ max	$\odot :=$ sum
$\otimes :=$ max	Worst case execution time of any single action	Total worst case execution time
$\otimes :=$ sum	–	Total execution time of all actions

Table 4.5.: Semantics of composition operators when cost equals power consumption.

cost ≈ power	$\odot :=$ max	$\odot :=$ sum
$\otimes :=$ max	Peak power consumption of any single action	–
$\otimes :=$ sum	Worst case peak power consumption	Worst case total power consumption

where undrilled objects should be painted. Formally speaking, the causal dependency between the two behavioral interfaces is not present in all use cases.

In such cases, modeling the desired order of execution should be performed in the plant model as opposed to the hardware model in order not to "contaminate" the (generic) hardware model with application-specific assumptions. Such a concept is currently not implemented, hence the only option to model order of execution until now is to specify explicit dependencies using predicates in the hardware model. A suitable implementation would be an extension of the plant model to allow specification of a total or relative order between processing operations on a work piece, for example $\mathsf{b}_{\mathsf{drill}}(\mathsf{a}_{\mathsf{obj}}) < \mathsf{b}_{\mathsf{paint}}(\mathsf{a}_{\mathsf{obj}})$. However, such ordering could depend on properties only known at runtime, for example: "If x_{wp1} is red, first drill and then mount. If x_{wp1} is black, first mount and then drill".

4.5. Summary

This chapter answers the question on how modular assembly lines and the tasks to be performed on them can be formally described in order to enable further automatic processing. For this purpose, we define a three-step approach, namely specification of *hardware model*, *plant model* and *task model*. Each step has different requirements on the background knowledge of the engineers that are responsible for the specification. Hence, the presented modeling workflow partitions the specification of the model according to different user roles.

An important aspect of the task model is that it does not specify the concrete steps to reach the production goal. Instead, it formally specifies the production goal itself. This approach intentionally forces an engineer to focus on the high-level outcome of the production task rather than the low-level implementation details required to implement it.

Although the formal specification seems quite tedious, the hierarchical structure of the models allows them to be reused: a hardware model for a specific type of MAL needs to be defined only once and can be reused in any plant model defined on top of it. Similarly, a plant model for a specific MAL only needs to be defined once, while an arbitrary number of task models can be defined for it.

Some deficits of the presented modeling language have also been identified; for example the tradeoff between the generality of the hardware model and the dependencies between the behavioral interfaces. The balancing of these parameters depends on the concrete application scenario.

The next chapter deals with how the formal description presented in this chapter is applied to automatic synthesis of control programs.

4.6. Related Work

Basile et al. state that in automation, "it is usual to start the control design from a description of the components of a system and of the interactions among them in terms of events, states and constraints" [BCDG08]. **Modeling of automation systems** as introduced in this chapter follows the same philosophy: system components are specified at the granularity of hardware modules, interactions are specified as behavioral interfaces with predicates that represent the state, while execution is bound to various preconditions that serve as constraints.

In order to allow intuitive modeling, some of the approaches for modeling automation systems have been achieved in **general purpose modeling languages** such as *Unified Modeling Language (UML)* [LLJ04]. However, "UML diagrams alone are not very useful as analysis, control synthesis or validation framework, especially [in] automation contexts" [BCDG08]. The reasons for this are that the various UML diagrams allow to model a system in many different ways and that they are not very precise with respect to formal analysis or verification.

Hence, automation processes are widely modeled using **Petri nets** [Mur89]. Reasons for their success are their "formal semantics, the graphical nature, the expressiveness, the availability of analysis techniques to prove structural properties (invariance properties, deadlock, liveness, etc.) and the possibility to define and evaluate performance indices (throughput, occupation rates, etc.)" [BCDG08]. Petri nets allow modeling "process synchronization, asynchronous events, concurrent operations, and conflicts or resource sharing for a variety of industrial automated systems at the discrete-event level" [Zho98]. Such modeling techniques have been widely applied to semiconductor manufacturing systems, for example in [Zho98, JCC97, KD97]. The strengths of Petri net based modeling of automation systems lie in the analysis. However, analysis "becomes impossible" for processes with "uncertainty due to resource contention, expected maintenance downtime, and unexpected failures". Hence, "simulation is a dominant approach to obtaining the performance data" [Zho98].

In order to be able to synthesize control programs, the approach considered in this work borrows some concepts from Petri nets, but applies them in a different context. For example, the model presented in this work considers work pieces and operating positions on which those work pieces may be located. Work pieces can be seen as tokens in a Petri net with the restriction that only one token is allowed per place (operating position). Transitions in Petri nets are represented by behavioral interfaces that define actions in order to change the state of the work pieces and to move them through the plant. Preconditions of the behavioral interfaces can be seen as enabling conditions for the transitions. However, not all information in the model is represented by tokens: a predicate model allows representing global knowledge by discrete values. Furthermore, the developed models feature additional information that is required in order to directly map a synthesized control program to the target platform. As opposed to Petri nets, these extensions adapt the presented approach to the automation domain. Furthermore, the two-player nature of the game modeled by the approach presented in this work models an automation tak more intuitively than a generic Petri net.

Capability-based modeling is common in order to describe the functionality a specific module is capable of. Keddis et al. demonstrated that the *information technology (IT)* systems for industrial automation can be automatically adapted based on a capability model of all stations in a plant [KKB13]. Their approach is based on a modular hardware platform and can even be extended to mobile robots. The approach in this work follows a similar approach: capabilities of the individual modules are encoded in behavioral interfaces and bound to certain preconditions to determine their availability. In contrast to Keddis et al., where material flow is calculated separately by adding transport operations manually, this work integrates material flow modeling directly into the software synthesis by modeling them as overlapping operating positions. This allows applying optimizations such as cost-bounded synthesis directly to processing *and* transport operations, which makes the approach more uniform.

CHAPTER 5

Industrial Control Program Generation Workflow

Contents

5.1. Approach **74**
5.2. Constraint Checking **77**
5.3. Model-to-model Transformation **79**
5.4. Model-to-text Transformation **86**
5.5. Game-based Solving **93**
5.6. Translation of Solver Output to Control Programs **96**
5.7. Discussion and Application to Running Example **100**
5.8. Summary **102**
5.9. Related Work **103**

Overview

The research question tackled in this chapter is the following:

> Given a formal description of the structure and capabilities of a *modular assembly line (MAL)* and the production task to achieve, how can (possibly decentralized) control programs be automatically synthesized for the *electronic control units (ECUs)* of the MAL?

This question is answered by defining a fully automatic process that transforms the formal description into a two-player game that serves as input to a suitable solver. The solver calculates a strategy such that one of the players wins the game if such a strategy exists. The resulting strategy in form of a finite state machine is then transformed into executable code for simulation or execution on the ECU(s).

5.1. Approach

In order to be able to apply game-based synthesis as introduced in Section 2.3, we transform the process model introduced in the previous chapter into a suitable input language for the game-based solver that represents the task to execute as a game between the two players Controller and Environment.

The available moves of the Controller correspond to the triggering of control commands on the ECU(s) in the plant, while the moves of Environment correspond to the observable (sensor) environment. Synthesizing a suitable control algorithm for Controller corresponds to a respective winning strategy in the two-player game. The solver reports that no winning strategy exists for Controller if the given task is infeasible.

The presented approach allows generating a single centralized control program or multiple decentralized control programs, one for each ECU. In the following, we compare the two approaches with each other.

5.1.1. Centralized Control Strategy

A *centralized control strategy* assumes that the full knowledge about the current (global) state is always available and that the invocation of all primitive control functions that implement the behavioral interfaces defined in the plant model is possible from a single program. If multiple ECUs are present in the plant, they are remotely controlled from that single program.

In this case, the information specified in the hardware model is used to connect the global control program to every single ECU in order to trigger the respective primitive control functions when appropriate. Figure 5.1 (a) illustrates this situation.

As indicated in Figure 5.1 (a), the program does not need to run on one of the ECUs in this case; it could instead be running on a development *personal computer (PC)*, for example. This setup has the advantage that debugging of the generated control program is easy, because no concurrency is involved.

5.1.2. Decentralized Control Strategy

In contrast, a *decentralized control strategy* consists of a dedicated control program per ECU. From a functional perspective, the parallel execution of all dedicated control programs has the same effects than the execution of the centralized control strategy. However, advantages of decentralized control include reduced communication overhead and hence the opportunity for tighter timing guarantees in the implementation, offering of modular "services" in the form of production steps that are carried out by the decentralized control programs as well as easier maintenance and exchange of individual control programs without affecting others. Since decentralized control programs are automatically derived from the centralized control strategy, there is no additional development effort involved, provided the respective target system is compatible to this approach – switching between centralized and decentralized execution can be performed simply by re-synthesis.

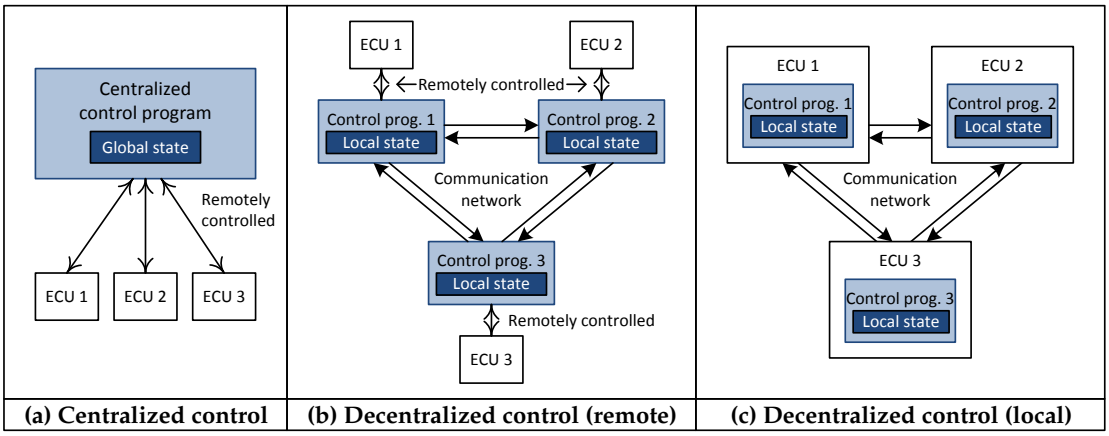


Figure 5.1.: Comparison of centralized and decentralized control strategies on an example with three ECUs: (a) centralized control program with remote control of all ECUs; (b) decentralized control programs with remote control of all ECUs; (c) decentralized control programs with local control of their associated ECU.

Dedicated control programs only have a limited view on the global system state and may trigger primitive control functions of devices that are connected to the respective control unit only. Figure 5.1 (b) shows a set of three decentralized control programs that remotely control one ECU each. Figure 5.1 (c) depicts the same setup, but this time the control program runs directly on the respective ECUs. Mixed forms are also possible.

In order to coordinate with the other ECUs specified in the hardware model, the dedicated control programs are connected to a communication network that allows to send values of local predicates and synchronization signals to other ECUs or to receive remotely available predicate values and to wait for synchronization signals. As indicated in Figure 5.1 (b) and (c), this work assumes bidirectional communication channels to be available between all ECUs in order to realize this requirement.

Information exchange in such a system is in general required if an ECU-local control program contains at least one sensor triggering, i.e., if the control program contains potential uncertainty. However, in the chosen approach, a transfer of the state between ECUs is required whenever the execution context switches between ECUs. The reason for this is that when a behavioral interface is executed, the predicate value updates are only applied in the control program that executed the behavioral interface; they are not "simulated" on the other ECUs. Implementing such a simulation would further reduce the required network bandwidth and is considered future work.

Algorithms 5.1 and 5.2 show the *anticipated* decentralized control programs for ECU instances $\hat{u}_{\mathsf{Plant}}$ and $\hat{u}_{\mathsf{Drill}}$ in task model $\mathfrak{T}_2$ (compare to the centralized control program shown in Algorithm 4.2 on page 69). This example uses the approach that is called "token-based replication of the full state" that is described in detail in Section 5.3.3.1. Some noteworthy properties of these programs are listed in the following:

- For decentralized control strategies, only one ECU is active at any point in time.

Algorithm 5.1: Anticipated decentralized version of Algorithm 4.2 for ECU $\hat{u}_{\text{Plant}}$.

```
1  set-current-ecu (x_û_Plant);
2  lever-push-û_Plant (x_wp1, π_L1.from, π_L1.to);
3  plate-rotate-û_Plant (x_wp1, π_RP1.p1, ...);
4  probe-height-û_Plant (x_wp1, π_H1.probe);
5  plate-rotate-û_Plant (x_wp1, π_RP1.p2, ...);
6  set-current-ecu (x_û_Drill);
7  transfer-state (x_û_Plant, x_û_Drill);
8  nop ();
9  nop ();
10 transfer-state (x_û_Drill, x_û_Plant);
11 plate-rotate-û_Plant (x_wp1, π_RP1.p3, ...);
12 lever-push-û_Plant (x_wp1, π_L2.from, π_L2.to);
```

Algorithm 5.2: Anticipated decentralized version of Algorithm 4.2 for ECU $\hat{u}_{\text{Drill}}$.

```
1  set-current-ecu (x_û_Plant);
2  nop ();
3  nop ();
4  nop ();
5  nop ();
6  nop ();
7  transfer-state (x_û_Plant, x_û_Drill);
8  if height-û_Drill (x_wp1, x_small) then
       drill-û_Drill (x_wp1, π_D1.drill);
   end
9  set-current-ecu (x_û_Plant);
10 transfer-state (x_û_Drill, x_û_Plant);
11 nop ();
12 nop ();
```

The set-current-ecu behavioral interface, for which the placement in the control program is automatically derived by the solver, is used to switch context from one ECU to another. In order to ensure that the solver picks the correct behavioral interfaces for a given ECU, the behavioral interfaces are replicated on all ECUs where they may be executed and a respective ECU-specific suffix is added to the behavioral interface name (e.g., lever-push-$\hat{u}_{\text{Plant}}$). For details, see Section 5.3.3.

- Additional guards in each step of the control program ensure that an ECU executes a no-operation (nop) if it is not the current ECU. The no-operation is non-blocking, i.e., the program immediately continues with the next statement.
- The transfer-state behavioral interface is used to transfer changes in the ECU-local state from the active ECU to another ECU. On the sending side (identified by the fact that the ECU the code is executed on corresponds to the first parameter), this operation ensures that the receiver is listening and sends the respective state update. On the receiving side (identified by the fact that the ECU the code is executed on corresponds to the second parameter), the operation blocks until a state update is received. A state update may contain an arbitrary number (including zero) of predicate updates, followed by a synchronization signal. Notice that the state transfer may include a new current ECU value, which is the reason why ECU $\hat{u}_{\text{Drill}}$ executes behavioral interfaces after receiving the state from $\hat{u}_{\text{Plant}}$ in line 7, where the new current ECU was determined as $\hat{u}_{\text{Drill}}$ in line 6.
- In this simple example, the transfer between ECUs is deterministic. In general, setting of the next active ECU could depend on sensor input. In the case that multiple ECUs might be the next active ECUs, a state update and synchronization signal has to be sent to all possible candidates.

5.1.3. Individual Steps of the Approach

The following list provides an overview of the steps carried out in order to obtain a synthesized (centralized or decentralized) control program from a process model. These steps are described in detail in the following sections.

1. **Constraint checking:** A set of check constraints is applied to the process model in order to ensure that only consistent models are used for game-based synthesis. This procedure reveals many design errors at an early stage of development and encourages confidence in the correct specification of the process model.
2. **Model-to-model transformation:** The model is subsequently extended and adapted in a *model-to-model transformation* in multiple steps. The transformation includes preparative steps in order to execute the resulting control program in a distributed way as introduced in the previous sections. Notice that the transformation does not modify the original model, instead the model specified by the user is cloned and the modifications carried out on the copy.
3. **Model-to-text transformation:** *Model-to-text transformation* converts the model into the input language for the solver, the *Planning Domain Definition Language (PDDL)*.
4. **Game-based solving:** The solver is applied to the PDDL specification and returns a *finite state machine (FSM)* of a suitable strategy if the specification is feasible or an error otherwise.
5. **Translation of solver output to control programs:** If the specification was feasible, the final step is to optimize and translate the finite state machine's textual representation into executable code[1].

5.2. Constraint Checking

Before translating the models into an input language for the solver, the consistency of the models is checked. For this purpose, we define a set of *check rules* $\mathfrak{C} = \{\mathfrak{c}_1, \mathfrak{c}_2, \ldots, \mathfrak{c}_\mathfrak{c}\}$ where each check rule $\mathfrak{c}_i, 1 \leq i \leq \mathfrak{c}$ is a function

$$\mathfrak{c}_i \colon \mathcal{M} \to \mathbb{B} \tag{5.1}$$

where $\mathcal{M}$ is the set of all possible process models $\mathfrak{M} = (\mathfrak{H}, \mathfrak{P}, \mathfrak{T})$.

Check Rule CR1 *At least one operating position per module type:*

$$\mathfrak{c}_1(\mathfrak{M}) = \begin{cases} 1 & \textit{if } \forall \varphi \in \Phi_\mathfrak{H} \colon \Pi_\varphi \neq \emptyset \\ 0 & \textit{otherwise} \end{cases} \tag{5.2}$$

Check Rule CR2 *Predicates are nullary, unary or binary:*

$$\mathfrak{c}_2(\mathfrak{M}) = \begin{cases} 1 & \textit{if } \forall v \in \mathcal{V}_\mathfrak{H} \colon a_v \in \{0, 1, 2\} \\ 0 & \textit{otherwise} \end{cases} \tag{5.3}$$

[1] Without loss of generality, the FSM is translated into C/C++ code in this work.

This check rule is currently required due to limitations in the underlying solver. It is not a limitation of the approach itself.

Check Rule CR3 *All modules in the plant model are topologically connected:*

$$\Theta_{\mathsf{L}_\mathfrak{P}} := \bigcup_{\ell=(\theta_s,\theta_r)\in\mathsf{L}_\mathfrak{P}} \{\theta_s, \theta_r\}$$

$$\mathfrak{c}_3(\mathfrak{M}) = \begin{cases} 1 & \textit{if } |\Psi_\mathfrak{P}| \leq 1 \vee \neg\exists\psi \in \Psi\colon \forall\pi \in \Pi_{\varphi_\psi}\colon \{\theta_\psi(\pi)\} \cap \Theta_{\mathsf{L}_\mathfrak{P}} = \emptyset \\ 0 & \textit{otherwise} \end{cases} \tag{5.4}$$

where $\Theta_{\mathsf{L}_\mathfrak{P}}$ is the set of all operating position instances referenced in the plant model, both source and replacement instances (compare Definition D13). The constraint ensures that either the number of modules is at most one or there exists no module instance ψ for which none of its operating position instances, expressed by $\theta_\psi(\pi)$, is in $\Theta_{\mathsf{L}_\mathfrak{P}}$.

Assuming we use the predicates "at" and "occupied" to represent the location of work pieces and the occupancy state of an operating position as introduced in Section 4.4.1, we add one more check rule to ensure the consistency between the two predicates:[2]

Check Rule CR4 *Consistent use of predicates "at" and "occupied"[3]:*

$$\mathfrak{c}_4(\mathfrak{M}) = \begin{cases} 1 & \textit{if } (\forall \mathsf{b} \in \mathcal{B}_\mathfrak{H}\colon \forall e \in \mathcal{E}_\mathsf{b}\colon \\ & \quad (e \equiv \mathsf{at}(w,\pi) \Rightarrow \exists e_2 \in \mathcal{E}_\mathsf{b}\colon e_2 \equiv \mathsf{occupied}(\pi)) \wedge \\ & \quad (e \equiv \neg\mathsf{at}(w,\pi) \Rightarrow \exists e_2 \in \mathcal{E}_\mathsf{b}\colon e_2 \equiv \neg\mathsf{occupied}(\pi)) \wedge \\ & \quad (e \equiv \mathsf{occupied}(\pi) \Rightarrow \exists e_2 \in \mathcal{E}_\mathsf{b}, w \in \mathcal{W}\colon e_2 \equiv \mathsf{at}(w,\pi)) \wedge \\ & \quad (e \equiv \neg\mathsf{occupied}(\pi) \Rightarrow \exists e_2 \in \mathcal{E}_\mathsf{b}, w \in \mathcal{W}\colon e_2 \equiv \neg\mathsf{at}(w,\pi)) \\ &) \wedge (\forall \mathsf{b} \in \mathcal{B}_\mathfrak{H}\colon \forall \hat{e} \in \widehat{\mathcal{E}}_\mathsf{b}\colon \\ & \quad \Big(\hat{e} \equiv \mathsf{at}(w,\pi) \Rightarrow \exists \hat{e}_2 \in \widehat{\mathcal{E}}_\mathsf{b}\colon \hat{e}_2 \equiv \mathsf{occupied}(\pi)\Big) \wedge \\ & \quad \Big(\hat{e} \equiv \neg\mathsf{at}(w,\pi) \Rightarrow \exists \hat{e}_2 \in \widehat{\mathcal{E}}_\mathsf{b}\colon \hat{e}_2 \equiv \neg\mathsf{occupied}(\pi)\Big) \wedge \\ & \quad \Big(\hat{e} \equiv \mathsf{occupied}(\pi) \Rightarrow \exists \hat{e}_2 \in \widehat{\mathcal{E}}_\mathsf{b}, w \in \mathcal{W}\colon \hat{e}_2 \equiv \mathsf{at}(w,\pi)\Big) \wedge \\ & \quad \Big(\hat{e} \equiv \neg\mathsf{occupied}(\pi) \Rightarrow \exists \hat{e}_2 \in \widehat{\mathcal{E}}_\mathsf{b}, w \in \mathcal{W}\colon \hat{e}_2 \equiv \neg\mathsf{at}(w,\pi)\Big) \\ &) \wedge (\forall c_i \in \mathcal{C}_{\mathcal{I}_\mathfrak{T}}\colon \\ & \quad (c_i \equiv \mathsf{at}(w,\pi) \Rightarrow \exists c_{i2} \in \mathcal{C}_{\mathcal{I}_\mathfrak{T}}\colon c_{i2} \equiv \mathsf{occupied}(\pi)) \wedge \\ & \quad (c_i \equiv \neg\mathsf{at}(w,\pi) \Rightarrow \exists c_{i2} \in \mathcal{C}_{\mathcal{I}_\mathfrak{T}}\colon c_{i2} \equiv \neg\mathsf{occupied}(\pi)) \wedge \\ & \quad (c_i \equiv \mathsf{occupied}(\pi) \Rightarrow \exists c_{i2} \in \mathcal{C}_{\mathcal{I}_\mathfrak{T}}, w \in \mathcal{W}\colon c_{i2} \equiv \mathsf{at}(w,\pi)) \wedge \\ & \quad (c_i \equiv \neg\mathsf{occupied}(\pi) \Rightarrow \exists c_{i2} \in \mathcal{C}_{\mathcal{I}_\mathfrak{T}}, w \in \mathcal{W}\colon c_{i2} \equiv \neg\mathsf{at}(w,\pi)) \\ &) \\ 0 & \textit{otherwise} \end{cases} \tag{5.5}$$

[2] For simplification reasons, we assume here that no $\mathcal{C}_{\mathcal{I}_\varphi}, \varphi \in \Phi_\mathfrak{H}$ contains any condition that deals with "at" or "occupied". This is a meaningful assumption, because it does not make sense that an initial module condition in the hardware model specifies the location of work pieces in the plant model.

[3] Some of the individual checks presented here only produce warnings as opposed to errors in the real implementation (MGSyn tool).

These constraints force the clearing of an operating position to be represented as an effect even if it is known to be filled by a different work piece in the same step. However, this is not a problem, because the effect lists are ordered and hence the subsequent filling of the same operating position can be specified at a subsequent item in the list.

Definition D16 (consistency) *A process model $\mathfrak{M}$ is called* consistent *if*

$$\forall 1 \leq i \leq \mathfrak{c} \colon \mathfrak{c}_i(\mathfrak{M}) = 1 \tag{5.6}$$

A model that is not consistent is called inconsistent.

If a model is inconsistent, a respective diagnostic message is reported to the user. Game-based synthesis is unavailable for inconsistent process models.

Notice that a specific implementation of a process model in form of a domain specific language requires the addition of many more check constraints. These checks are necessary, because implementing a real *domain-specific language (DSL)* requires the addition of artifacts such as unique objects names. In contrast, uniqueness of objects is guaranteed by the kind of notation in the presented strict mathematical formalism. Hence, the MGSyn tool presented in Section 7.1 implements more than 100 checks for detection of duplicate object names, ambiguities and correct referencing of types and objects across elements of the DSL.

Provided a process model $\mathfrak{M}$ is consistent, we apply model-to-model transformation as follows.

5.3. Model-to-model Transformation

5.3.1. Addition of Sensor Response Actions

The input model contains behavioral interfaces for Controller actions (*actuations*) and Environment actions (*sensor triggerings*). However, a sensor triggering does not model a move of the Environment. Instead, it primarily models the fact that Controller passes the control over the game to Environment. The actual turn of Environment and its properties are not yet explicitly represented. However, this information is required in order to properly set up the game in the game-based solver.

Algorithm 5.3 shows the pseudocode of an algorithm that extends the model accordingly. The algorithm consists of three steps.

In the first step, a *sensor response* is created for every sensor triggering. The sensor response has the same properties as its sensor triggering with the following exceptions:

- The response is only executable when it is not Controller's turn (¬P0TRAN) and the predicate b-active is true, which is specifically created to represent whether Controller wants to know the sensor input from exactly the given sensor[4].

[4]This implies that we can handle only one instance of a certain type of sensor per plant model in centralized control program synthesis and one instance per ECU in decentralized control program synthesis. This limitation can be removed by introducing a sensor-instance specific parameter to b-active.

- The effects of the response are that the next turn is carried out by Controller (P0TRAN) and the sensor activation predicate is reset ($\neg$b-active).
- Without loss of generality, cost of sensor responses is always zero; the cost of sensor responses is represented instead in the respective sensor triggering.

In the second part of the algorithm, we modify the existing actions such that:

- All actions (except the not yet added sensor responses) are Controller moves (P0TRAN).
- All sensor triggerings cause the next turn to be taken by Environment and activate the respective sensor response.

This way, we force the environment to always answer Controller's sensor triggerings with a respective response. The newly created sensor responses are added to the set of available actions in the last step.

5.3.2. Unique Identification of Module Instances

The control program from Algorithm 4.1 on page 69 contains two module instances of type lever. Line 1 triggers L1, while on line line 6, L2 is activated. This can be easily seen by inspecting the positional parameters passed to lever-push. However, it is undesirable for the lever's *primitive control function (PCF)* to interpret those parameters in order to decide the module instance to trigger, because it requires topological information about the plant to be present in the PCFs. An elegant solution for this problem is to let the solver explicitly choose the instance to trigger. For this purpose, we extend the model as follows.

1. **Addition of device and behavioral objects:** A new object type $\mathcal{X}_{\text{device}}$ is introduced with objects $\{x_\psi \mid \psi \in \Psi_{\mathfrak{P}}\}$ (i.e., one object for every module instance) and a new object type $\mathcal{X}_{\text{behavioral}}$ with objects $\{x_{\mathsf{b}} \mid \mathsf{b} \in \mathcal{B}_\varphi, \varphi \in \Phi_{\mathfrak{H}}\}$ (i.e., one object for every behavioral interface).
2. **Addition of module-specific predicates and initial conditions:** A predicate device-hold: $\mathcal{X}_{\text{device}} \times \Pi \to \mathbb{B}$ is added that represents the fact that a module instance "owns" a set of operating positions. For initialization, add for every operating position of a module instance $\psi \in \Psi_{\mathfrak{P}}$ a set of initial conditions $\mathcal{C}_{\mathcal{I}_{\text{device-hold}_\psi}} := \{\text{device-hold}(x_\psi, x_\pi) \mid \psi \in \Psi_{\mathfrak{P}} \wedge \pi \in \Pi_\psi\}$.

   Furthermore, we add a predicate device-use: $\mathcal{X}_{\text{device}} \times \mathcal{X}_{\text{behavioral}} \to \mathbb{B}$ that indicates which behavioral interfaces are defined for a module type. For initialization, we add for every module instance $\psi \in \Psi_{\mathfrak{P}}$ a set of initial conditions $\mathcal{C}_{\mathcal{I}_{\text{device-use}_\psi}} := \{\text{device-use}(x_\psi, x_{\mathsf{b}}) \mid \psi \in \Psi_{\mathfrak{P}} \wedge \mathsf{b} \in \mathcal{B}_{\varphi_\psi}\}$.
3. **Adaptation of behavioral interfaces:** In order to let the game-based solver determine the correct module instance to trigger, add a parameter named x_{device} with domain $\mathcal{X}_{\text{device}}$ to every behavioral interface b. In order to constrain the possible valuations for x_{device}, we add a precondition device-hold$(x_{\text{device}}, \mathsf{a}_{\text{pos}})$ for every *positional* argument $\mathsf{a}_{\text{pos}} \in \mathsf{A}_{\mathsf{b}}$ (i.e., for every argument of type Π). Furthermore, we add a precondition device-use$(x_{\text{device}}, x_{\mathsf{b}})$, where x_{b} is the object that corresponds to b.

Algorithm 5.3: Pseudo-code for addition of sensor responses to the process model.

Input: Process model $\mathfrak{M} = (\mathfrak{H}, \mathfrak{P}, \mathfrak{T})$
Output: Updated process model $\mathfrak{M}$

```
// First pass: create sensor responses
1  B+ := ∅;                                  // set of newly added sensor responses
2  foreach behavioral interface b ∈ B_P do
3      if R_b ≠ ∅ then                                  // if b is sensor triggering
4          V_P := V_P ∪ {b-active};  // b-active → 𝔹 indicates pending triggering
5          A_{b_r} := A_b;              // response has same arguments as triggering
6          C_{P b_r} := C_{P b} ∪ {¬P0TRAN, b-active};  // precondition: pending trig.
7          E_{b_r} := E_b ∪ {P0TRAN, ¬b-active};   // deactivate pending triggering
8          Ê_{b_r} := ∅;                                  // no conditional effects
9          R_{b_r} := R_b;   // same sensor response conditions as triggering
10         p̂_{b_r} := p̂_b;                // apply control function from triggering
11         η_{b_r} := 0;          // actual cost represented in sensor triggering
12         b_r := (A_{b_r}, C_{P b_r}, E_{b_r}, Ê_{b_r}, R_{b_r}, p̂_{b_r}, η_{b_r}, p̈_{b_r});      // new sensor response
13         B+ := B+ ∪ {b_r};                          // schedule b_r for addition
14         foreach module type φ ∈ Φ_H do
15             if b ∈ B_φ then
16                 B_φ := B_φ ∪ {b_r};              // add the sensor response to the
17                 break;              // module that owns the sensor triggering
18             end
19         end
20     end
21 end
// Second pass: update original actions
22 foreach behavioral interface b ∈ B_P do
23     C_{P b} := C_{P b} ∪ {P0TRAN};                // ensure Controller takes turns
24     if R_b ≠ ∅ then                                    // if b is sensor triggering
25         E_b := E_b ∪ {¬P0TRAN, b-active};        // enable env. sensor response
26         R_b := ∅;                                // no sensor response conditions
27         p̂_b := ⊥;              // remove control function from triggering,
                                                  // now triggered in sensor response
28     end
29 end
// Add new sensor responses to process model
30 B_P := B_P ∪ B+;
```

This has two effects: on the one hand, the device argument has to be chosen such that it represents the module instance with the correct set of operating positions. For example, if a plant model contains multiple lever module instances with disjoint operating positions, then the interface lever-push will have its device parameter set to the device that owns the operating position where the work piece is initially located[5]. On the other hand, we demand that the device parameter is chosen such that the respective module type owns the respective behavioral interface. Consider for example two levers, where L1 pushes a work piece from operating position A to B and L2 from B to A and assume further that two different behavioral interfaces are used for pushing forth and back. Then due to the device-use related precondition, the device argument of the respective behavioral interfaces will be correctly set to either L1 or L2 depending on the push direction.

5.3.3. Replication of Model Elements for Distributed Execution

A control engineer may select to generate a centralized or a decentralized control strategy for a given plant model. If distributed control is applied, i.e., a dedicated control program is to be generated for every ECU in the plant model, some constraints need to be added to the plant model that encode the fact that the system state is not completely observable from the point of view of a specific ECU.

The approach targets the following behavior: all ECUs start with the same initial state as specified in the model. The individual control programs are executed in parallel with suitable synchronization points. Whenever the triggering of a primitive control function depends on a predicate whose value may be influenced by behavioral primitives that are executed on another ECU, ensure that the respective value is first transferred to the ECU in question. Whenever such a "replicated" predicate is modified, make sure that the outdated values present on remote ECUs are not used anymore.

The following three approaches to realize these constraints have been analyzed and some of them implemented as part of this work.

- **Token-based replication at granularity of predicates:** This approach transfers the state based on the individual predicates. For every predicate, a single token exists in the distributed setup that determines which ECU owns the current value of the predicate. Before reading or writing a predicate, its token needs to be acquired, possibly implying a transfer action.

  This approach is scalable with respect to network bandwidth at the cost of a higher number of variables during game-based synthesis. Unfortunately, detailed analysis shows that this approach leads to a high synthesis time and the decentralized control programs are not feasible for execution if certain conditions apply. In particular, this approach only networks the predicate values that are either part of the preconditions or (conditional) effects of the actions being ex-

[5] Notice that we assume that modules with at least one behavioral interface without positional arguments to be instantiated only once; otherwise the instance to trigger could not be decided when the strategy is executed. For example, consider two signal lamps with interface set-state($\mathcal{X}_{\text{state}}$) and $\mathcal{X}_{\text{state}} = \{x_{\text{red}}, x_{\text{yellow}}, x_{\text{green}}\}$; then the valuation of the device parameter would not be constrained.

ecuted. However, the guards for action execution most probably contain other predicates as well, whose values are not networked. Hence, this approach is not feasible. Nevertheless, Appendix A depicts the algorithm in detail.

- **Token-based replication of the full state:** This approach uses just a single token to determine which ECU owns the current "image" of all state predicates. Before reading or writing any predicate, the token needs to be acquired, possibly implying a transfer action.

  This approach is less modular than the first approach. Nevertheless, it allows for a much shorter synthesis time and the synthesized control programs do not suffer from the problem mentioned above.
- **Pessimistic replication analysis:** This approach completely omits the construction of transfer actions in the game-based solver. Instead, an analysis is performed on the resulting control strategy in order to determine when it is required to transfer a predicate value. The current implementation is a heuristic approach with pessimistic replication: predicate values are transferred if there is at least one control flow where the predicate value is unknown on the receiving side when that predicate is part of a guard expression (e.g., a precondition) or a (conditional) effect. As such, this approach has the advantage that it is performed at predicate granularity while not adversely influencing synthesis time.

The following sections describe the latter two approaches in detail.

5.3.3.1. Token-based Replication of the Full State

In order to implement token-based replication of the full state, an automatic model transformation is used to adapt the plant model. Algorithm 5.4 shows the pseudocode for this transformation. Line numbers in the following refer to that algorithm.

1. **Addition of ECU objects:** In order to represent which ECU is currently active, we add for every ECU instance $\hat{u} \in \widehat{\mathcal{U}}_{\mathfrak{P}}$ a corresponding object $x_{\hat{u}}$ (lines 1 to 2).
2. **Representation of token:** A predicate $\mathsf{current\text{-}ecu} : \mathcal{X}_{\mathsf{ECU}} \to \mathbb{B}$ is added that represents which ECU currently owns the token and hence the current image of all state predicates (line 3). This predicate is true for at most one ECU at a time[6]. The initial state is "all false", which means that no owner for the token has been determined yet.
3. **Splitting of behavioral interfaces:** For every behavioral interface, we determine whether it needs to be executed on different ECUs, and if yes, clone the behavioral interface (lines 4 to 13). This ensures that the game-based solver indicates on which ECU the action is to be triggered by choosing the respective variant. For example, if behavioral interface lever-push is required on ECUs e1 and e2 (because each ECU controls at least one module of type lever), then two ECU-local behavioral interfaces named lever-push-e1 and lever-push-e2 replace the original behavioral interface. Assuming behavioral interface drill is only required on ECU e1, then a single behavioral interface drill-e1 replaces the existing one.

[6]Notice that this predicate can profit from binary encoding as introduced later in Section 5.5.3.2.

Algorithm 5.4: Pseudo-code for token-based replication of the full state.

Input: Process model $\mathfrak{M} = (\mathfrak{H}, \mathfrak{P}, \mathfrak{T})$
Output: Updated process model $\mathfrak{M}$

// Addition of ECU objects
1 $\mathcal{X}_{\mathsf{ECU}} := \{x_{\mathsf{e}_{\hat{u}}} \mid \hat{u} \in \widehat{\mathcal{U}}_{\mathfrak{P}}\}$; // set of ECU objects
2 $\mathcal{X}_{\mathfrak{P}} := \mathcal{X}_{\mathfrak{P}} \cup \mathcal{X}_{\mathsf{ECU}}$; // add set of ECU objects to plant model

// Representation of token
3 $\mathcal{V}_{\mathfrak{H}} := \mathcal{V}_{\mathfrak{H}} \cup \{\mathsf{current\text{-}ecu}\}$;

// Splitting of behavioral interfaces
4 $\mathcal{B}^{+} := \emptyset$; // set of newly added behavioral interfaces
5 $\mathcal{B}^{-} := \emptyset$; // set of behavioral interfaces to remove
6 **foreach** *behavioral interface* $\mathsf{b} \in \mathcal{B}$ **do**
7 **foreach** *ECU instance* $\hat{u} \in \widehat{\mathcal{U}}_{\mathfrak{P}}$ **do**
8 **if** $\mathsf{b} \notin \mathcal{B}_{\varphi_{\psi}}$ *($\hat{u}$ does not control a module with behavioral interface* b*)* **then continue**;
9 define new behavioral interface $\mathsf{b}_{\hat{u}} := \mathsf{b}$ and assign to same module type as b;
10 $\mathcal{B}^{+} := \mathcal{B}^{+} \cup \{\mathsf{b}_{\hat{u}}\}$; // schedule $\mathsf{b}_{\hat{u}}$ for addition
11 $\mathcal{B}^{-} := \mathcal{B}^{-} \cup \{\mathsf{b}\}$; // schedule b for removal
12 **end**
13 **end**

// Addition of token-related behavioral interfaces
14 $\mathsf{b}_{\mathsf{set\text{-}current\text{-}ecu}} := ((\mathsf{a}_{\mathsf{ecu}} \in \mathcal{X}_{\mathsf{ECU}}), \mathcal{C}_{\mathcal{P}\mathsf{b}_{\mathsf{set\text{-}current\text{-}ecu}}}, \mathcal{E}_{\mathsf{b}_{\mathsf{set\text{-}current\text{-}ecu}}}, \emptyset, \emptyset, \bot, 0, 0)$;
15 $\mathcal{C}_{\mathcal{P}\mathsf{b}_{\mathsf{set\text{-}current\text{-}ecu}}} := \{\neg\mathsf{current\text{-}ecu}(x_{\hat{u}}) \mid \hat{u} \in \widehat{\mathcal{U}}_{\mathfrak{P}}\}$;
16 $\mathcal{E}_{\mathsf{b}_{\mathsf{set\text{-}current\text{-}ecu}}} := \{\mathsf{current\text{-}ecu}(\mathsf{a}_{\mathsf{ecu}})\}$;
17 $\mathsf{b}_{\mathsf{transfer\text{-}state}} := ((\mathsf{a}_{\mathsf{source}}, \mathsf{a}_{\mathsf{dest}} \in \mathcal{X}_{\mathsf{ECU}}), \mathcal{C}_{\mathcal{P}\mathsf{b}_{\mathsf{transfer\text{-}state}}}, \mathcal{E}_{\mathsf{b}_{\mathsf{transfer\text{-}state}}}, \emptyset, \emptyset, \hat{p}_{\mathsf{transfer\text{-}state}}, 0, 0)$;
18 $\mathcal{C}_{\mathcal{P}\mathsf{b}_{\mathsf{transfer\text{-}state}}} := \{\mathsf{current\text{-}ecu}(\mathsf{a}_{\mathsf{source}})\}$;
19 $\mathcal{E}_{\mathsf{b}_{\mathsf{transfer\text{-}state}}} := \{\neg\mathsf{current\text{-}ecu}(\mathsf{a}_{\mathsf{source}}), \mathsf{current\text{-}ecu}(\mathsf{a}_{\mathsf{dest}})\}$;
20 $\mathcal{B}^{+} := \mathcal{B}^{+} \cup \{\mathsf{b}_{\mathsf{set\text{-}current\text{-}ecu}}, \mathsf{b}_{\mathsf{transfer\text{-}state}}\}$; // schedule for addition

// Update of set of behavioral interfaces
21 **foreach** *module type* $\varphi_{\mathfrak{H}}$ **do**
22 $\mathcal{B}_{\varphi} := \mathcal{B}_{\varphi} \setminus \mathcal{B}^{-}$;
23 **end**
24 $\mathcal{B} := (\mathcal{B} \setminus \mathcal{B}^{-}) \cup \mathcal{B}^{+}$;

4. **Addition of token-related behavioral interfaces:** The last step is to add two additional behavioral interfaces, one to determine the initial owner of the token and one to transfer the token (along with an image of the current state) from one ECU to another (lines 14 to 20). set-current-ecu($\mathsf{a}_{\mathsf{ecu}}$) is executed to designate $\mathsf{a}_{\mathsf{ecu}}$ as the initial owner of the token. transfer-state($\mathsf{a}_{\mathsf{source}}, \mathsf{a}_{\mathsf{dest}}$) is executed to transfer the token from ECU $\mathsf{a}_{\mathsf{source}}$ to ECU $\mathsf{a}_{\mathsf{dest}}$. In general, the availability of such transfer actions between a pair of ECUs depends on the network topology available. A fully connected topology is assumed in this work.

   The primitive control function $\hat{p}_{\mathsf{transfer\text{-}state}}$ implements the actual state transfer between the two ECUs. It is automatically provided by code generation from the model and has the semantics described in Section 5.1.2. Notice that the cost for both operations is set to zero here in order not to affect quantitative synthesis. In case transfer operations are expensive, the cost may be set to a value larger than zero in order to instruct the solver to reduce the number of transfer operations to a minimum. Without loss of generality, $\mathsf{b}_{\mathsf{set\text{-}current\text{-}ecu}}$ and $\mathsf{b}_{\mathsf{transfer\text{-}state}}$ are added to the set of behavioral interfaces of the first module in the plant model (not shown in Algorithm 5.4).

5.3.3.2. Pessimistic Replication Analysis

Another possibility for replication of predicate values is based on a pessimistic analysis of the generated strategy. This technique combines the fine-granular replication from the token-based approach at predicate granularity with fast synthesis time. In the following, we sketch a suitable algorithm. The basic idea is to simulate all possible execution traces of the control program and insert transfer operations in the execution sequence where they are required in at least one execution trace. Notice that unlike the previous algorithm, which harness the power of the game-based solver to find a suitable solution, this one is a post-processing step applied to the generated strategy.

For every ECU $\hat{u} \in \widehat{\mathcal{U}}_{\mathfrak{P}}$, let $\mathcal{V}_u(\hat{u})$ denote the set of unknown predicates in the ECU-specific control program, i.e., those predicates for which the respective control program does *not* "know" the current value. The initial state is globally known, hence the sets are initially empty. The following algorithm is then applied to all blocks in the strategy:

1. **Determination of target ECU:** First, determine on which ECU $\hat{u}_e$ the current action is to be executed.
2. **Required transfer operations for guard condition and (conditional) effects:** Determine the set of predicates $\mathcal{V}_g$ that are tested in the current guard condition (corresponding to the preconditions of the action to be executed) as well as the set of predicates $\mathcal{V}_e$ written in the (conditional) effects of the respective action. Let $\mathcal{V}_m = \mathcal{V}_u(\hat{u}_e) \cap (\mathcal{V}_g \cup \mathcal{V}_e)$ denote the missing predicates for execution of the current action. Transfer the predicate value for all $v \in \mathcal{V}_m$ from an arbitrary ECU $\hat{u}_{know}$ for which $v \notin \mathcal{V}_u(\hat{u}_{know})$. Such an ECU is guaranteed to exist. Then set $\mathcal{V}_u(\hat{u}_e) := \mathcal{V}_u(\hat{u}_e) \setminus \mathcal{V}_m$.
3. **Action execution and predicate invalidation:** On ECU $\hat{u}_e$, evaluate the guard condition in order to determine whether to execute the action. If the action is

executed, apply the (conditional) effects of the action to the local copies of the predicates; those predicates are all contained in $\mathcal{V}_e$.

On all other ECUs $\hat{u} \in \widehat{\mathcal{U}} \setminus \{\hat{u}_e\}$, we need to assume the worst case, which is that the guard condition evaluates to true on $\hat{u}_e$. Hence, we need to invalidate the local copies of the potentially modified predicates on these ECUs: set $\mathcal{V}_u(\hat{u}) := \mathcal{V}_u(\hat{u}) \cup \mathcal{V}_e$ for all $\hat{u} \in \widehat{\mathcal{U}} \setminus \{\hat{u}_e\}$.

This algorithm statically inserts transfer operations between ECUs based on a safe over-approximation of the situations where a transfer is required.

5.4. Model-to-text Transformation

This work uses PDDL to encode specifications for passing them to a solver. In the following section, a brief summary of PDDL is given.

5.4.1. Planning Domain Definition Language

The *planning domain definition language* (PDDL) was originally designed for the AIPS-98 planning competition [GHK+98]. It combines features from previously existing languages such as ADL [Ped89] and UMCP [EHN94]. "PDDL is intended to express the "physics" of a domain, i.e., what predicates there are, what actions are possible [...] and what the effects of actions are" [GHK+98]. Since the original introduction, PDDL has been extended in many ways [GL05, HET+06]. The *Backus-Naur form (BNF)* of the most recent version 3.1 of PDDL is available in [Kov11]. A PDDL specification consists of two definitions:

- The *domain specification* (compare Listing 5.1) usually defines requirements, types, constants, (parameterized) predicates and actions. Actions are guarded by preconditions over predicates that must hold in order for the action to be executable. Actions define effects in terms of updates to the predicate values. Effects may also depend on predicate values (so-called conditional effects).
- The *problem specification* (compare Listing 5.2) defines the initial state and goal state for a concrete planning problem. States are specified in terms of the predicates defined in the domain specification.

Listings 5.1 and 5.2 show examples for PDDL domain and problem specifications. Listing 5.1 defines a domain called stackDomain (line 1) that allows us to reason about locations and objects (line 4). For this purpose, a suitable solver must implement certain requirements (line 2). The predicates $\text{loc}\colon \text{object} \times \text{location} \to \mathbb{B}$ (object is at location) and $\text{on}\colon \text{object} \times \text{object} \to \mathbb{B}$ (first object is on top of second object) are used to represent the state of the system (lines 5f). Action $\text{put-obj-on}\colon \text{object} \times \text{object} \times \text{location}$ defined in lines 7ff specifies that an object can be stacked onto another object if they are at the same location and neither of the objects is stacked or has an object stacked on it. Action $\text{put-obj-off}\colon \text{object} \times \text{object}$ unstacks an object (lines 22ff) and action $\text{move-obj}\colon \text{object} \times \text{location} \times \text{location}$ moves an object including the object stacked on it, if any (lines 27ff).

Listing 5.1: Example of a PDDL domain specification

```
(define (domain stackDomain)
        (:requirements :strips :typing :conditional-effects
                       :negative-preconditions)
        (:types location object)
        (:predicates (loc ?obj - object ?at - location)
                     (on ?top ?bot - object) )
        (:action put-obj-on
                 :parameters (?obj ?newbot - object ?at - location)
                 :precondition (and
                    ;; ?obj and ?newbot is at location ?at
                    (loc ?obj ?at) (loc ?newbot ?at)
                    ;; ?obj not stacked itself
                    (forall (?bot - object) (not (on ?obj ?bot)))
                    ;; nothing stacked on ?obj
                    (forall (?top - object) (not (on ?top ?obj)))
                    ;; ?newtop not stacked
                    (forall (?bot - object) (not (on ?newbot ?bot)))
                    ;; nothing stacked on ?newtop
                    (forall (?top - object) (not (on ?top ?newbot))) )
                 :effect (on ?obj ?newbot)
        )
        (:action put-obj-off
                 :parameters (?obj ?oldbot - object)
                 :precondition (on ?obj ?oldbot)
                 :effect (not (on ?obj ?oldbot))
        )
        (:action move-obj
                 :parameters (?obj - object ?from ?to - location)
                 :precondition (loc ?obj ?from)
                 :effect (and (loc ?obj ?to) (not (loc ?obj ?from))
                              (forall (?top)
                                      (when (on ?top ?obj)
                                            (and (not (loc ?top ?from))
                                                 (loc ?top ?to) )
                                      )
                              )
                 )
        )
)
```

Listing 5.2: Example of a PDDL problem specification

```
(define (problem stackAndMove)
        (:domain stackDomain)
        (:objects loc1 loc2 - location obj1 obj2 - object)
        (:init (loc obj1 loc1) (loc obj2 loc2))
        (:goal (and (loc obj1 loc2) (on obj2 obj1)))
)
```

The problem specification stackAndMove (Listing 5.2) states which object and locations are available (line 42), what their initial configuration is (line 43) and defines a goal for the solver (line 44). The output of a solver for such kinds of problems is typically of form of an execution trace of actions such that the goal state is reached after the last action in the trace has been executed. A solver might come up with one of the following execution traces:

- move−obj(obj1, loc1, loc2), put−obj−on(obj2, obj1)
- move−obj(obj2, loc2, loc1), put−obj−on(obj2, obj1), move−obj(obj1, loc1, loc2)

Notice that many more (most probably uninteresting) traces are possible, because the effects of put−obj−on and put−obj−off cancel each other out.

This simple example is based on perfect information, i.e., no uncertainty is present. When problems with imperfect information need to be handled, actions are introduced that retrieve the required information during execution using conditional effects. In this case, the solver has to make sure that the goal is reached in all possible execution traces. This means that the output is not a sequence of parameterized actions, but an FSM, in which the transitions depend on the trace of actions already executed. Instead of storing the execution trace, the transition conditions are typically formulated over the current state, i.e., the values of the predicates.

Assume for example that a hostile environment might decide to let a stacked object fall down. Without loss of generality, assume that this might only happen directly after the object has been stacked onto another object and that this might only happen once[7] during the execution of the control program. In this case, a suitable control program has to check the valuation of predicate "on" after an execution of put-on to ensure that the state is as expected.

The *Game Arena Visualization and Synthesis Plus! (GAVS+)* solver used in this work uses a subset of PDDL as input language and produces a suitable FSM as output for games with incomplete information if a solution exists. It is introduced in the following.

5.4.2. GAVS+

Game arena visualization and synthesis (GAVS) [CBLK10] and the extended version *GAVS+* [CKLB11] are software frameworks for solving and visualizing games typically encountered in theoretical computer science. While *Game Arena Visualization and Synthesis (GAVS)* focuses on turn-based two-player games on finite graphs, GAVS+ greatly extends the set of supported game types and offers better interoperability with other tools [Che12]. Both frameworks consist of a *graphical user interface (GUI)* and a set of synthesis engines for different types of games, among them reachability games, Büchi games and parity games.

The original motivation for creating GAVS was that many problems from theoretical computer science can be encoded as two-player games and solving these games corre-

[7] This restriction ensures that no infinite loop of stacking and falling down is possible. By fine-tuning the maximum number of allowed hostile actions, we can express how tolerant the control program should be with respect to these uncertainties. The more likely it is for the stacked object to fall down, the higher the number of tolerated hostile actions should be.

sponds to proving properties of the respective problems. The graphical user interface of GAVS allows step-by-step solving and visualization of games and hence fosters the understanding of the respective algorithms. It also allows steepening the learning curve in teaching the respective algorithms. Refer to [Che12] for more information.

In this work, we use the game solving capability of GAVS+ to synthesize industrial control programs from PDDL specifications. For this purpose, we first need to translate the *process model* $\mathfrak{P} = (\mathfrak{H}, \mathfrak{P}, \mathfrak{T})$ introduced in Sections 4.2 and 4.3 into PDDL.

Notice that the PDDL domain and problem specification could be written manually and directly used as input for the solver.However, maintaining the process model in this form is tedious and error-prone. Generating the PDDL files from a suitable model has the advantage that a model is easier to maintain and due to its high level of abstraction, it can be directly validated against the check constrains defined in Section 5.2 before any subsequent steps are performed.

5.4.3. Translation of Process Model to PDDL Domain Specification

In the following, we describe how a consistent process model $\mathfrak{M} = (\mathfrak{H}, \mathfrak{P}, \mathfrak{T})$ consisting of a *hardware model* $\mathfrak{H} = (\Phi_{\mathfrak{H}}, \mathcal{V}_{\mathfrak{H}}, \mathcal{X}_{\mathfrak{H}}, \mathcal{U}_{\mathfrak{H}})$, a suitable *plant model* $\mathfrak{P} = (\mathfrak{H}, \mathcal{X}_{\mathfrak{P}}, \Psi_{\mathfrak{P}}, \widehat{\mathcal{U}}_{\mathfrak{P}}, \mathsf{L}_{\mathfrak{P}})$ and a respective *task model* $\mathfrak{T} = (\mathfrak{P}, \mathcal{C}_{\mathcal{I}\mathfrak{T}}, \sigma_{\mathfrak{T}}, g_{\mathfrak{T}}, \eta_{\max\mathfrak{T}}, \odot_{\mathfrak{T}}, \otimes_{\mathfrak{T}}, d_{\mathfrak{T}})$ are translated into a PDDL domain specification.

5.4.3.1. PDDL Requirements Section

The following fixed list of flags is listed in the :**requirements** field of the PDDL domain:

- :**strips**: Allows use of STRIPS [NF70, FN72] semantics. This means that a suitable planner must understand concepts of conditions, operations, initial state and goal state, i.e., interpret the keywords :**predicates**, :**action**, :**objects**, :**init** and :**goal**.
- :**typing**: Allows use of :**types** keyword for defining variable types and use of type names in declaration of variables. In addition to meaningful grouping of objects, this allows to restrict the valuation of predicate and action parameters.
- :**negative−preconditions**: Allows to use negated preconditions for actions. For example, this is required for environment actions, which have the precondition ¬P0TRAN (see Section 5.4.3.4).
- :**disjunctive−preconditions**: Allows "**or**" in :**goal** and :**preconditions** terms.
- :**conditional−effects**: Allows to use "**when**" as an operator in action effects. Used to represent conditional effects of behavioral interfaces.
- :**equality**: Supports "=" (equality operator) as built-in predicate. Required for comparison of parameter values in preconditions of parallelized actions[8].
- :**fluents**: Allows function definitions and use of effects with assignement operators and numeric preconditions. Required for quantitative synthesis[9].

[8] May be omitted if $\sigma_{\mathfrak{T}} \notin \{\sigma_{\mathrm{rp}}, \sigma_{\mathrm{rcp}}\}$.
[9] May be omitted if $\sigma_{\mathfrak{T}} \notin \{\sigma_{\mathrm{rc}}, \sigma_{\mathrm{rcp}}\}$.

5.4.3.2. PDDL Types Section

For every *object type* $\mathcal{X}_i$ from the set of objects $\mathcal{X}_{\mathfrak{H}} \cup \mathcal{X}_{\mathfrak{P}}$, add an item with name i to the :**types** section of the PDDL domain. This allows to use the types in parameters of predicates and actions.

5.4.3.3. PDDL Constants Section

For all objects $x \in \mathcal{X}_{\mathfrak{H}} \cup \mathcal{X}_{\mathfrak{P}}$, add an item to the :**constants** section of the PDDL domain, along with the respective type. This allows to refer to those objects in parameters of predicates and actions.

5.4.3.4. PDDL Predicates Section

For all predicates $v \in \mathcal{V}_{\mathfrak{H}}$, add an item to the :**predicates** section of the PDDL domain, listing the parameters and their types accordingly. If multiple types are allowed, use the "either" keyword, for example :**parameters** ?param − (either type1 type2).

The additional zero-argument predicate P0TRAN is added to represent which player takes turns next: if P0TRAN is 1, the it is *player 0*'s turn (i.e., Controller), otherwise the next turn is performed by *player 1* (i.e., Environment). Uncontrollable non-determinism is assumed (with respect to the available environment moves) when P0TRAN is 0.

5.4.3.5. PDDL Functions Section

If quantitative synthesis is considered (i.e., $\eta_{\max\mathfrak{T}} < \infty$), a :**functions** block is added to the PDDL domain specification as follows: (:**functions** (total-cost)) The total-cost function is a so-called *fluent* that is used to represent the accumulated cost of the invoked actions. In contrast to normal (Boolean) predicates, the value of a fluent is an integer number. The name of this function is recognized by the solver and treated specially in order to only synthesize strategies that do not violate the cost bound in case a quantitative solver is selected. Within the solver, total-cost is encoded as a predicate and the number of state variables increases accordingly.

5.4.3.6. PDDL Actions Section

Single actions:

For every behavioral interface $\mathsf{b} \in \mathcal{B}$, where $\mathcal{B} = \bigcup_{\varphi \in \Phi_{\mathfrak{H}}} B_\varphi$, add an :**action** section to the PDDL domain as follows:

- The name of the action is derived from the name of the behavioral interface.
- :**parameters** are derived from the ordered argument list $\mathsf{A_b}$ (including $\mathsf{a_{device}}$).
- :**precondition** is derived from the behavioral interface preconditions $\mathcal{C}_{\mathcal{P}_\mathsf{b}}$. For actions that represent Environment moves, the additional precondition ¬P0TRAN is added. For all other actions, the additional precondition P0TRAN is added.

- :**effect** is derived from the list of unconditional and conditional behavioral interface effects $\mathcal{E}_{\mathsf{b}}$ and $\widehat{\mathcal{E}}_{\mathsf{b}}$. For sensor triggerings, set POTRAN to 0, meaning an environment action must follow. For environment actions, set POTRAN back to 1 in order to let the control program continue with the next action. Conditional effects are represented using the "**when**" operator. In case of quantitative synthesis (i.e., $\eta_{\max} < \infty$), the following statement is added to the effects of the action if $\eta_{\mathsf{b}} > 0$: (increase (total-cost) η_{b}) The "increase" effect is available when ":**fluents**" is used as a requirement. It increases the value of the provided fluent by η_{b} [GL13].

Parallelized actions:

In case of parallel action execution (compare R12 on page 54), i.e., when the *degree of parallelization* d is larger than one[10] and a solver $\sigma_{\mathfrak{T}} \in \{\sigma_{\mathrm{rp}}, \sigma_{\mathrm{rcp}}\}$ is selected, then we need to add actions representing parallel execution. Given the set of behavioral interfaces $\mathcal{B}$ and a degree of parallelization $d > 1$, then actions from set $\mathcal{B}_{\|d\|} = \bigcup_{1<i\leq d} \mathcal{B}^i$ with $\mathcal{B}^i = \underbrace{\mathcal{B} \times \mathcal{B} \times \ldots}_{i \text{ times}}$ are automatically generated with the following restrictions:

1. $(\mathsf{b}_1, \ldots, \mathsf{b}_d)$ contains at most one Environment action (i.e., sensor triggering)[11].
2. No parallelized action has already been generated for a different permutation of $(\mathsf{b}_1, \ldots, \mathsf{b}_d)$.
3. All behavioral interfaces $(\mathsf{b}_1, \ldots, \mathsf{b}_d) \in \mathcal{B}^i$ are parallelizable, i.e., $\forall 1 \leq i \leq d$: $\ddot{p}_{\mathsf{b}_i} = 1$. This flag may be set to 0 in situations where:
   a) Parallelization is not possible due to reasons not represented in the model (e.g., due to physical constraints in the real plant).
   b) Parallelization is known to not be of any benefit (to reduce synthesis time).
   c) Technical limitations of the synthesis engine do not allow parallelization. This is for example the case when multiple behavioral interfaces with conditional effects are combined.
4. None of the effects $\mathcal{E}_{\mathsf{b}_i}$ and conditional effects $\widehat{\mathcal{E}}_{\mathsf{b}_i}, 1 \leq i \leq d$ conflict with each other, i.e. no concurrent updates with different valuation are performed on any predicates. This work implements a heuristic to detect such situations in conjunction with the fact that none of the arguments to a parallelized behavioral interface may be identical (see below).

An element of the set $\mathcal{B}^i$ is denoted as $(\mathsf{b}_0 \| \mathsf{b}_1 \| \ldots \| \mathsf{b}_{i-1})$. The properties of such a parallelized actions are derived as follows:

- The name of the parallelized action is a concatenation of the names of the individual actions and a prefix.
- :**parameters** is the concatenated list of parameters from the underlying behavioral interfaces. Let $\mathsf{A}_{\mathsf{b}} = (\mathcal{X}_{\mathsf{b}_0}, \mathcal{X}_{\mathsf{b}_1}, \ldots, \mathcal{X}_{\mathsf{b}_{b-1}})$ denote the list of *argument domains* of behavioral interface $\mathsf{b} \in \bigcup_{\varphi \in \Phi_{\mathfrak{H}}} \mathcal{B}_\varphi$ according to Definition D6 on page 49. Furthermore, let $\mathsf{A}_{(\mathsf{b}_0 \| \mathsf{b}_1 \| \ldots \| \mathsf{b}_{i-1})} = (\mathcal{X}_{\mathsf{b}_{00}}, \mathcal{X}_{\mathsf{b}_{01}}, \ldots, \mathcal{X}_{\mathsf{b}_{10}}, \mathcal{X}_{\mathsf{b}_{11}}, \ldots)$ denote the con-

[10] Currently only $d \in \{1, 2, 3\}$ is supported, where 1 means no parallelization.
[11] An option exists to enable parallelization of multiple sensor triggerings, but is not fully supported.

catenated list of the argument domains of the parallelized action of degree i over the behavioral interfaces $\mathsf{b}_0, \mathsf{b}_1, \ldots, \mathsf{b}_{i-1}$. Then the :**parameters** section is filled with elements from $\mathsf{A}_{(\mathsf{b}_0\|\mathsf{b}_1\|\ldots\|\mathsf{b}_{i-1})}$ and their respective types. A numeric suffix j indicating the number $1 \leq j \leq i$ of the behavioral interface the parameter was derived from is appended to the parameter names to ensure uniqueness.

- :**precondition**: A parallelized action is eligible for execution only if no concurrent update on predicates takes place. Such situations are safely underestimated by assuming that a parallel action is eligible for execution if the underlying behavioral interfaces are invoked with mutually exclusive parameters. This runtime assumption is sufficient in combination with the static restrictions for parallel execution from above. Rewrite $\mathsf{A}_{(\mathsf{b}_0\|\mathsf{b}_1\|\ldots\|\mathsf{b}_{i-1})}$ as $(\mathcal{X}_{(\mathsf{b}_0\|\mathsf{b}_1\|\ldots\|\mathsf{b}_{-1}i)_0}, \mathcal{X}_{(\mathsf{b}_0\|\mathsf{b}_1\|\ldots\|\mathsf{b}_{i-1})_1}, \ldots, \mathcal{X}_{(\mathsf{b}_0\|\mathsf{b}_1\|\ldots\|\mathsf{b}_{i-1})_{a-1}})$, where a is the total number of arguments. Then we need to ensure that the following holds[12]:

$$\forall j, k \colon \mathcal{X}_{(\mathsf{b}_0\|\mathsf{b}_1\|\ldots\|\mathsf{b}_{i-1})_j} \neq \mathcal{X}_{(\mathsf{b}_0\|\mathsf{b}_1\|\ldots\|\mathsf{b}_{i-1})_k} \vee j = k \tag{5.7}$$

  These constrains are expressed in the precondition of the parallelized action in a list combined by conjunctions of the form (**not** (= a_{p_1} a_{p_2})). In addition, the preconditions $\mathcal{C}_{\mathcal{P}_{\mathsf{b}_i}}$ of the underlying behavioral interfaces $\mathsf{b}_0, \mathsf{b}_1, \ldots, \mathsf{b}_{i-1}$ of the parallelized action are added as conjunctions.

- :**effect**: The effect of the parallelized action is the conjunction of the effects of the behavioral interfaces $\mathsf{b}_0, \mathsf{b}_1, \ldots, \mathsf{b}_{i-1}$. The guards in the precondition protect from concurrent updates on the predicates. In case of quantitative synthesis (i.e., $\sigma_{\mathfrak{T}} \in \{\sigma_{\mathrm{rc}}, \sigma_{\mathrm{rcp}}\}$), the cost of the parallelized action $(\mathsf{b}_0\|\mathsf{b}_1\|\ldots\|\mathsf{b}_{i-1})$ is calculated statically (i.e., at generation time of the PDDL domain) by evaluating $\bigotimes_{0 \leq j < i-1} \eta_{\mathsf{b}_j}$.

As an example, consider action $\mathsf{b}_{\text{lever-push}}(x_{\text{dev}}, x_{\text{wp}}, \pi_{\text{from}}, \pi_{\text{to}})$ of the lever, which allows to use module instance x_{dev} to move a work piece x_{wp} from operating position π_{from} to operating position π_{to}, and action $\mathsf{b}_{\text{drill}}(x_{\text{dev}}, x_{\text{wp}}, \pi_{\text{pos}})$ of the drill, which allows to drill work piece x_{wp} at operating position π_{pos}. For parallel execution, action $\mathsf{b}_{\text{par\_lever-push\_lever-push}}(x_{\text{dev}_1}, x_{\text{wp}_1}, \pi_{\text{from}_1}, \pi_{\text{to}_1}, x_{\text{dev}_2}, x_{\text{wp}_2}, \pi_{\text{from}_2}, \pi_{\text{to}_2})$ is automatically derived for moving two different work pieces on two different levers at the same time. The preconditions of this action include $x_{\text{dev}_1} \neq x_{\text{dev}_2}$, $x_{\text{wp}_1} \neq x_{\text{wp}_2}$, $\pi_{\text{from}_1} \neq \pi_{\text{from}_2}$, $\pi_{\text{from}_1} \neq \pi_{\text{to}_2}$, $\pi_{\text{to}_1} \neq \pi_{\text{from}_2}$ and $\pi_{\text{to}_1} \neq \pi_{\text{to}_2}$ to ensure that parameter values are different[13]. Furthermore, an action $\mathsf{b}_{\text{par\_lever-push\_drill}}(x_{\text{dev}_1}, x_{\text{wp}_1}, \pi_{\text{from}_1}, \pi_{\text{to}_1}, x_{\text{dev}_2}, x_{\text{wp}_2}, \pi_{\text{pos}_2})$ is derived for concurrent pushing of one work piece and drilling of another. Notice that an action $\mathsf{b}_{\text{par\_drill\_lever-push}}$ is not added, because a different permutation of $\mathsf{b}_{\text{drill}}$ and $\mathsf{b}_{\text{lever-push}}$ has already been added.

In case of quantitative synthesis, the cost of the parallelized actions is calculated as follows[14]: assume $\otimes := \mathsf{sum}$ and action $\mathsf{b}_{\text{lever-push}}$ has cost 3, then parallelized action $\mathsf{b}_{\text{par\_lever-push\_lever-push}}$ has cost $3 \otimes 3 = 6$. Likewise, if $\otimes := \mathsf{max}$, then total cost is 3.

[12] Actually we just need to ensure that parameters of the same type have different values. The tool developed along with this thesis honors this fact. For reasons of simplicity, we omit it here and give an example below where comparisons are listed only for parameters with matching types.

[13] Constraints such as $x_{\text{dev}_1} \neq \pi_{\text{from}_2}$ as not generated, because x_{dev_1} and π_{from_2} have different types.

[14] Notice that since $\otimes_{\mathfrak{T}}$ should be commutative, the order of input parameters does not matter.

5.4.4. Translation of Process Model to PDDL Problem Specification

In the following, we describe how relevant parts of the given hardware model $\mathfrak{H}$, plant model $\mathfrak{P}$ and task model $\mathfrak{T}$ are translated into a PDDL problem specification.

5.4.4.1. Initial State

The :**init** list of the PDDL problem specification is derived from all initial conditions of the module instances in the plant model and the initial conditions in the task model. Consider the set $\mathcal{C}_\mathcal{I}$ of all initial conditions:

$$\mathcal{C}_\mathcal{I} = \left(\bigcup_{\varphi \in \Phi_\mathfrak{H}} \mathcal{C}_{\mathcal{I}_\varphi} \right) \cup \mathcal{C}_{\mathcal{I}_\mathfrak{T}} \tag{5.8}$$

For every element in $\mathcal{C}_\mathcal{I}$, an item is added to the :**init** list in the PDDL problem specification. All unlisted parameter combinations of predicates are considered to be 0.

5.4.4.2. Goal State

The goal specification $g_\mathfrak{T}$ in the task model is directly interpreted as reachability condition in the game. It is hence put as :**goal** condition in the PDDL problem specification.

According to the definition of PDDL, all predicates that are not explicitly referred to in the goal specification can take an arbitrary value. This matches the expected semantics in the game-based solver.

5.4.4.3. Selection of Appropriate Solver

The type of solver $\sigma_\mathfrak{T}$ is directly passed to the underlying synthesis engine as a parameter when it is invoked. Hence, it does not need to be represented in the PDDL domain or problem specification.

5.5. Game-based Solving

This section provides a cursory overview of the GAVS+ solver used in the context of this work. Notice that the respective implementation is not part of this work and was originally designed in the context of [Che12, CJG+11].

5.5.1. Purpose of GAVS+

GAVS+ is an open source software framework that allows to "visualize and solve some of the most common two-player games encountered in theoretical computer science" [Che12]. It focuses on two-player turn-based games on finite graphs. GAVS+ solves many types of games (e.g., reachability, safety, Büchi, (weak) parity, Staiger-Wagner, Muller, Streett). For details, please refer to [Che12].

5.5.2. Restrictions of GAVS+

Currently, GAVS+ does not support quantification over objects [Che12]. Quantifications need to be rewritten as enumerations of concrete objects in the problem. Since this is possible in all cases, this constraint does not limit the expressiveness of the solver input language.

Furthermore, the game-based extension of GAVS+ currently fully supports predicates with at most two parameters only (compare Check Rule CR2 on page 77). Consequently, predicates with more parameters have to be manually split and consistently treated in initial state specifications, preconditions and effects. For example, consider the predicate moved(a_{obj}, a_{from}, a_{to}) indicating that an object a_{obj} has moved from location a_{from} to location a_{to}. Then we can split this predicate into three by introducing a new type of object $\mathcal{X}_{key}$ and three predicates moved-obj(a_{key}, a_{obj}), moved-from(a_{key}, a_{from}) and moved-to(a_{key}, a_{to}) with $a_{key} \in \mathcal{X}_{key}$. Although this notation is tedious, it does not limit the expressiveness of the solver input language.

5.5.3. Optimizations for Game-based Synthesis

The complexity of game-based synthesis in a discrete arena is dominated by the number of states, which in turn depends on the number of Boolean variables in the domain.

Game solving in GAVS+ is based on a symbolic representation of the state transition system. A state in this system is a specific valuation of the predicates. Consider the domain $\mathcal{X} = \mathcal{X}_{\mathfrak{H}} \cup \mathcal{X}_{\mathfrak{P}}$ of all objects and the set $\mathcal{C_P}^i$ of all predicates of arity i. Assuming the worst case, which is that all predicate parameters range over all objects in $\mathcal{X}$, a state of the system is an element of the following tuple [CJG+11][15]:

$$\mathbb{B}^{|\mathcal{C_P}^0|} \times \mathbb{B}^{|\mathcal{C_P}^1| \cdot |\mathcal{X}|} \times \cdots \times \mathbb{B}^{|\mathcal{C_P}^x| \cdot |\mathcal{X}|^x} \tag{5.9}$$

A high number of variables leads to the *state explosion problem*: every added Boolean variable doubles the number of possible states. Even with a small number of variables, this can lead to a surprisingly large number of states that makes it impossible to solve the problem in acceptable time, which is usually within the range of seconds to minutes on common PC hardware (compare Chapter 7).

In order to reduce game solving time, a set of optimization techniques is applied that identifies relevant regions of the game graph defined by the state transition system (so-called subarenas or local games) that are sufficient to solve the game. The following paragraphs summarize some of these optimizations.

[15] Notice that the approach presented in this work is currently limited to predicates with at most two parameters, i.e., $\forall i > 2 \colon |\mathcal{C_P}^i| = 0$. Nevertheless, the number of states is huge for typical values of $|\mathcal{X}|$.

5.5.3.1. Constant Replacement

Some predicates in the state transition system may never change, because they are not part of any effect of any action in the domain. Hence, they can be replaced by their constant value (depending on the initial state) in the state transition system [CJG+11].

5.5.3.2. Binary Encoding

For some predicates $v \in \mathcal{V}$ in the state transition system, the invariant holds that they are true for at most one combination of parameters. The predicate can be represented by $\lceil \log_2(n+1) \rceil$ instead of n variables in this case[16]. This approach is called *binary encoding*. Binary encoding is currently implemented for predicates with two parameters for concrete values of the first parameter and for predicates with one parameter.

The $\mathsf{at}(\mathsf{a}_{\mathsf{obj}}, \mathsf{a}_{\mathsf{pos}})$ predicate from Section 4.4.1.2 is a good example: an object $\mathsf{a}_{\mathsf{obj}}$ may only be located at (at most) one operating position. If seven locations are possible for the object, only 3 instead of 7 Boolean variables are required to represent the state[17].

Whether a predicate can be binary encoded or not is automatically determined from the initial state and the potential modifications applied to the predicates in the effects of behavioral interfaces. A heuristic that considers actions where preconditions and effects are represented as conjunction of (negated) predicates is presented in [CJG+11].

5.5.3.3. Goal-indifferent Variable Analysis

Modifications to some variables in the state transition system may be identified as irrelevant to reaching the goal: whenever there exists a winning strategy that modifies the variables, there exists another winning strategy that does not modify them. A heuristic is based on a recursive backward analysis of the predicates and their valuation in the preconditions of actions whose effects lead to the set of winning states. When goal-indifferent variables are identified, actions whose preconditions and effects only contain these variables may be completely omitted [CJG+11].

5.5.3.4. Cost Bound Exceeding Path Elimination

In quantitative synthesis, the sequential composition operator $\odot$ is used to calculate the total cost of a play while it is being constructed[18]. As soon as the calculated total cost for the current play exceeds the cost bound $\eta_{\max}$, the respective branch can be eliminated, because it does not lead to a feasible strategy.

[16] $\lceil \log_2(n) \rceil$ is not enough, because the default state "all false" needs to be represented as well.

[17] Notice that the state must be represented separately for every valuation of the first parameter $\mathsf{a}_{\mathsf{obj}}$. For example, if $\mathcal{W} = \{x_{\mathsf{wp1}}, x_{\mathsf{wp2}}\}$, then 6 Boolean variables would be required instead of 14.

[18] Notice that the implementation of $\odot$ is part of the synthesis algorithm and hence we know the concrete total cost value during construction.

5.6. Translation of Solver Output to Control Programs

In case the specification is feasible, the output from the solver is an FSM represented in a "Java-like controller program" [Che12]. Even if distributed synthesis is enabled, the output is still a single FSM. The following sections describe the structure of the output and how ECU-local control programs are automatically derived if distributed synthesis is used.

Grammar 5.1 denotes the BNF of the output. This kind of output can be translated into a C/C++ program by using suitable text-based transformations as described in the following.

5.6.1. Optimization of Guard Conditions

Due to the nature of the solver, the guards typically contain checks for predicate values that can be statically calculated from the initial state and the potential updates happening in any control flow of the program. Hence, we first optimize the guard conditions in the generated strategy in order to remove "superfluous" tests. A superfluous test is a condition whose value is guaranteed to be known as true or false.

There are two major reasons why this optimization is performed: on the one hand, eliminating guard conditions that do not influence the semantics of the control program increases its readability. On the other hand, this technique allows reducing the complexity of the analysis in the following steps, which for example reduces the communication required between ECUs in distributed synthesis.

The transformation approach used is an heuristic approach with complexity $\mathcal{O}(n)$, where n is the number of blocks. For example, the guard to the action lever-push-e1 (...) on line 2 of Algorithm 5.1 on page 76 actually contains the expression depicted in Algorithm 5.5 when output by the solver.

Algorithm 5.5: Actual guard conditions generated by solver for Algorithm 5.1.

```
1 if ... ∧ at(x_wp1, π_L1.from) ∧ occupied(π_L1.from) ∧ ... then
2 |   lever-push-e1 (x_wp1, π_L1.from, π_L1.to);
3 end
4 ...
```

Since this is the first action performed, work piece x_{wp1} must be located at operating position $\pi_{\text{L1.from}}$, because the set of initial conditions specifies this state. Hence, both terms are replaced by the constant "true". Repeated application of this technique leads to all guard conditions that appear before the first sensor triggering to be replaced by the constant "true", because the state is deterministically decidable until that point.

A suitable algorithm for predicates with up to two parameters is described in the following. Let $\mathcal{S}$ denote the set of potentially true predicate valuations.

- A predicate $v \to \mathbb{B}$ with zero parameters is either not contained in $\mathcal{S}$ or $\mathcal{S}$ contains either "$v = 1$" or "$v \in \{0, 1\}$", where the latter means that the value may be true or false. The fact that v is not present in $\mathcal{S}$ represents a guaranteed false value.

⟨*output*⟩ ::= ⟨*block*⟩*

⟨*block*⟩ ::= ⟨*block-header*⟩ ⟨*transition*⟩+ ⟨*block-footer*⟩

⟨*block-header*⟩ ::= '`/* Start of block `' ⟨*digit*⟩+ '`: */`'
'`/* ==================== */`'

⟨*block-footer*⟩ ::= '`/* End of block `' ⟨*digit*⟩+ '`: */`'

⟨*transition*⟩ ::= '`if (`' ⟨*guard*⟩ '`) {`' ⟨*action*⟩ '`}`'

⟨*guard*⟩ ::= ⟨*clauses*⟩

⟨*clauses*⟩ ::= ⟨*clause*⟩ ['` || `' ⟨*clauses*⟩]

⟨*clause*⟩ ::= '`( `' ⟨*expressions*⟩ '` )`'

⟨*expressions*⟩ ::= ⟨*expression*⟩ ['` && `' ⟨*expressions*⟩]

⟨*expression*⟩ ::= ['`!`'] ⟨*term*⟩

⟨*term*⟩ ::= ⟨*predicate-0*⟩
| ⟨*predicate-1*⟩ '`(`' ⟨*identifier*⟩ '`)`'
| '`(`' ⟨*predicate-1-binary*⟩ '`==`' ⟨*identifier*⟩ '`)`'
| ⟨*predicate-2*⟩ '`(`' ⟨*identifier*⟩ '`,`' ⟨*identifier*⟩ '`)`'

⟨*predicate-0*⟩ ::= ⟨*identifier*⟩ of zero-argument predicate

⟨*predicate-1*⟩ ::= ⟨*identifier*⟩ of one-argument predicate

⟨*predicate-1-binary*⟩ ::= ⟨*identifier*⟩ of one-argument predicate with binary encoding

⟨*predicate-2*⟩ ::= ⟨*identifier*⟩ of two-argument predicate

⟨*action*⟩ ::= ⟨*action-name*⟩ '`(`' ⟨*action-params*⟩ '`)`';

⟨*action-name*⟩ ::= ⟨*identifier*⟩ of behavioral interface whose primitive control function is to be executed

⟨*action-params*⟩ ::= ⟨*identifier*⟩ ['`, `' ⟨*action-params*⟩]

⟨*identifier*⟩ ::= ⟨*letter*⟩ ⟨*literal*⟩*

⟨*literal*⟩ ::= ⟨*letter*⟩ | ⟨*digit*⟩

⟨*letter*⟩ ::= '`A`' | ... | '`Z`' | '`a`' | ... | '`z`' | '`_`' | '`#`'

⟨*digit*⟩ ::= '`0`' | '`1`' | ... | '`9`'

Grammar 5.1: Simplified *Backus-Naur form (BNF)* of output produced by the GAVS+ solver: the resulting FSM is encoded in imperative form. The root node ⟨*output*⟩ consists of a sequence of blocks, where the nth block represents the nth Controller move in the corresponding game. Each block contains the possible state transitions guarded by "`if`" statements, each one with a condition over the predicates that encode the respective game state. The condition is built from a disjunction of conjunctions (i.e., it is in disjunctive normal form). The triggered primitive control functions, identified by ⟨*action-name*⟩, update the state space internally.

- A predicate $v\colon \mathcal{X} \to \mathbb{B}$ with one parameter is not contained in $\mathcal{S}$ or $\mathcal{S}$ contains either "$v = x$" with $x \in \mathcal{X}$ or "$v \in \mathcal{V}_\mathcal{S}$" with $\mathcal{V}_\mathcal{S} \subseteq \mathcal{X}$, where the latter means that the value is any of the ones in the given set. The fact that v is not present in $\mathcal{S}$ represents a guaranteed "all false" value.
- A predicate $v\colon \mathcal{X}_1 \times \mathcal{X}_2 \to \mathbb{B}$ with two parameters is either not contained in $\mathcal{S}$ or $\mathcal{S}$ contains either "$v(x_1) = x_2$" with $x_2 \in \mathcal{X}_2$ or "$v(x_1) \in \mathcal{V}_\mathcal{S}$" with $\mathcal{V}_\mathcal{S} \subseteq \mathcal{X}_2$ for $x_1 \in \mathcal{X}_1$, where the latter means that the value is any of the ones in the given set. The fact that $v(x_1)$ is not present in $\mathcal{S}$ represents a guaranteed "all false" value for a first parameter valuation of x_1.

$\mathcal{S}$ is initialized with the guaranteed valuations from the initial conditions in the hardware model $\mathcal{C}_{\mathcal{I}_\varphi}$ for $\varphi \in \Phi_\mathfrak{H}$ and the task model $\mathcal{C}_{\mathcal{I}_\mathfrak{T}}$. Then the following algorithm is applied for every block in the synthesized strategy:

1. **Encoding of terms in guard condition:** For every guard condition, inspect every individual term that tests for a valuation of a predicate. Let $(\neg)v(\dots)$ denote the predicate with the valuation being tested (where $\neg$ indicates a negated term), then:
   a) If v has zero parameters, the term is negated and "$v = 1$" $\in \mathcal{S}$, the term evaluates to false and hence the whole guard condition as well.
   b) If v has zero parameters, the term is negated and v is not at all contained in $\mathcal{S}$ (i.e., guaranteed to be false), the term evaluates to true and may be omitted.
   c) If v has zero parameters, the term is not negated and "$v = 1$" $\in \mathcal{S}$, the term evaluates to true and may be omitted.
   d) If v has zero parameters, the term is not negated and v is not at all contained in $\mathcal{S}$ (i.e., guaranteed to be false), the term evaluates to false and hence the whole guard condition as well.
   e) If $v(\dots, x_x)$ has at least one parameter with last parameter x_x, the term is negated and either $v(\dots)$ is not at all contained in $\mathcal{S}$ (i.e., guaranteed to be false) or "$v(\dots) = x'$" $\in \mathcal{S}$ with $x_x \neq x'$ or "$v(\dots) \in \mathcal{X}'$" $\in \mathcal{S}$ with $x_x \notin \mathcal{X}'$, the term evaluates to true and may be omitted.
   f) If $v(\dots, x_x)$ has at least one parameter with last parameter x_x, the term is negated and "$v(\dots) = x_x$" $\in \mathcal{S}$, the term evaluates to false and hence the whole guard condition as well.
   g) If $v(\dots, x_x)$ has at least one parameter with last parameter x_x, the term is not negated and either $v(\dots)$ is not at all contained in $\mathcal{S}$ (i.e., guaranteed to be false) or "$v(\dots) = x'$" $\in \mathcal{S}$ with $x_x \neq x'$ or "$v(\dots) \in \mathcal{X}'$" $\in \mathcal{S}$ with $x_x \notin \mathcal{X}'$, the term evaluates to false and hence the whole guard condition as well.
   h) If $v(\dots, x_x)$ has at least one parameter with last parameter x_x, the term is not negated and "$v(\dots) = x_x$" $\in \mathcal{S}$, the term evaluates to true and may be omitted.

   Otherwise the term *may* evaluate to true and is hence retained.
2. **State update:** If the guard condition is guaranteed to evaluate to true, apply the effects of the executed behavioral interface directly to $\mathcal{S}$ (i.e., remove negated

predicate valuations from $\mathcal{S}$ and add non-negated valuations). If the guard condition *may* be true, apply the effects of the executed behavioral interface as potential updates as follows. Let $(\neg)v(\ldots, x_x)$ denote the effect under consideration, then:

- If the term is negated and v is not at all contained in $\mathcal{S}$, do nothing.
- If the term is negated and "$v(\ldots) = x$" $\in \mathcal{S}$ with $x \neq x_x$, do nothing.
- If the term is negated and "$v(\ldots) = x_x$" $\in \mathcal{S}$, remove it from $\mathcal{S}$.
- If the term is negated and "$v(\ldots) \in \mathcal{X}$" $\in \mathcal{S}$, remove x_x from $\mathcal{X}$ if it exists. Replace "$v(\ldots) \in \mathcal{X}$" with "$v(\ldots) = x$" if only one element x remains in $\mathcal{X}$.
- If the term is not negated and v is not contained at all in $\mathcal{S}$, add "$v(\ldots) = x_x$" to $\mathcal{S}$.
- If the term is not negated and "$v(\ldots) = x_x$" $\in \mathcal{S}$, do nothing.
- If the term is not negated and "$v(\ldots) = x$" $\in \mathcal{S}$ with $x \neq x_x$, replace it with "$v(\ldots) \in \{x, x_x\}$".
- If the term is not negated and "$v(\ldots) \in \mathcal{X}$" $\in \mathcal{S}$, add x_x to $\mathcal{X}$ if it is not yet contained.

This algorithm is currently implemented in MGSyn as an external post-processing step to optimize the output strategy. For this purpose, a so-called annotation file is generated during model transformation that contains a summary of all required information as an additional input to the external post-processing step.

5.6.2. Implementation of Inter-ECU Synchronization

In order to illustrate how synchronization and state transfer is implemented between ECUs, we assume token-based replication of the full state being used as described in Section 5.3.3.1. In this case, transfer-state ($x_{\hat{u}_{\text{source}}}$, $x_{\hat{u}_{\text{dest}}}$) actions are synthesized by the solver at points where control flow needs to switch from ECU $x_{\hat{u}_{\text{source}}}$ to ECU $x_{\hat{u}_{\text{dest}}}$ as illustrated in Algorithms 5.1 and 5.2 on page 76. Although the state transfer itself is implemented by the PCF $\hat{p}_{\text{transfer-state}}$ of transfer-state as described in Section 5.3.3.1, this action needs special treatment.

In order to explain why, consider a control program where the guard condition of the transfer-state action on $x_{\hat{u}_{\text{source}}}$ depends on sensor input on that ECU that has been acquired since the last transfer of the control flow to $x_{\hat{u}_{\text{source}}}$. In this case, the execution of $\hat{p}_{\text{transfer-state}}$ on $x_{\hat{u}_{\text{source}}}$ is not guaranteed and hence the other ECUs cannot "know" whether or not it will happen. However, the control program on $x_{\hat{u}_{\text{dest}}}$ *must* wait for a synchronization signal (and a potential state transfer) independent of its current state, because that state is potentially outdated.

In order to resolve this problem, the transfer-state action is treated specially: the automatic post-processing ensures that on $x_{\hat{u}_{\text{source}}}$, we send the synchronization signal independent of whether the guard condition evaluates to true or not. If a state transfer follows, because the guard condition was true and $\hat{p}_{\text{transfer-state}}$ was actually executed, this signals $x_{\hat{u}_{\text{dest}}}$ to continue execution of its control program. Otherwise, the local value of the current-ecu predicate on $x_{\hat{u}_{\text{dest}}}$ still tells the control program that another ECU is

currently active and hence it does not execute any actions until the next invocation of transfer-state or until the control program finishes, whichever happens first.

This algorithm synchronizes the two ECUs specified by the parameters of a transfer-state action at any point where a state transfer *might* be necessary. Notice that ECUs that are not referred to by the parameters of the transfer-state action are not affected by this post-processing step.

5.6.3. Addition of ECU-specific Guards

Since we split the behavioral interfaces before passing them to the solver (compare Section 5.3.3), it is guaranteed that each action invocation corresponds to a specific ECU. In order to derive local control programs for every ECU, we determine for every $\langle action \rangle$ on which ECU it is to be executed. Then, we add an additional global conjunction to the guard of the respective block that tests whether the "current ECU" (i.e., the one the control program is executed on) corresponds to the one that should execute the action.

For example, the command lever-push-e1 (...) shown on line 2 of Algorithm 5.1 is actually of the following form after this step:

Algorithm 5.6: ECU-specific guards generated by solver for Algorithm 5.1.

```
1 if currentEcu == e1 then
2 |  lever-push-e1 (x_wp1, π_L1.from, π_L1.to);
3 end
```

This also explains the no-operations in Algorithms 5.1 and 5.2: the above condition simply evaluates to false on the respective ECU. In this way, we derive individual control programs for every ECU from the output generated by the solver.

5.6.4. Cleanup

Some final steps are carried out to obtain a valid C/C++ program: for more concise code, each block is split into a separate C/C++ function. A function strategy() is added that calls the individual block functions in sequence. Finally, all characters that may not be part of identifiers in C/C++ programs are replaced by underscore characters. The resulting code is used during compilation of the executable program for both simulation and execution on the real hardware.

5.7. Discussion and Application to Running Example

The presented approach generates optimized (not optimal) control programs for a given *process model*. Tracking of cost in quantitative synthesis allows "forbidding" solutions that are not acceptable with respect to metrics such as execution time or power consumption. Complex examples with quantitative synthesis are provided in Chapter 7. When decentralized control programs are synthesized, the process model is

transformed in order to represent the constraints that are present during decentralized control program execution. This approach allows switching between centralized and decentralized execution. Since both the centralized and decentralized control program generation use the same input and output interfaces to communication with the solver, the solver can in principle be exchanged by a different game-based algorithm.

With few exceptions, the terminology used in the generated control program corresponds to the names of the elements in the model. This makes the generated control programs easily understandable by humans. The fact that only a single control program is generated for distributed synthesis (which is run with different parameters depending on which ECU it represents) avoids the need to reconstruct the control flow across different source files or functions during debugging.

However, the suggested approach has a few downsides that are listed in the following. The most significant problem is that when the specification is infeasible (i.e., the synthesis engine does not find a control strategy), no counter-example or partial execution trace is provided to indicate the cause. A manual review of the model and the intermediate steps is required in such cases. Countermeasures such as counter-example generation would need to be implemented in the solver.

Another issue is that the structural complexity of the control program increases with the number of sensor inputs, because more sensor inputs require more complex guard expressions to determine whether a behavioral interface should be executed or not. Furthermore, due to the fact that the contained FSM is sequentially encoded in the control program, the synthesis result consists of an "overlap" of multiple execution traces. This is a limitation of the solver currently being used. Consider for example the task specification "Drill undrilled work pieces" from Section 4.4.3.2. While Algorithm 4.2 on page 69 shows the expected control program in the ideal case, a simplified version of the actually generated centralized control program is shown in Algorithm 5.7. On the right side, the so-called block numbers are indicated as output by the game-based solver. They correspond to the 1-based number of the Controller move being executed. Notice that in the algorithm, the triggering of all behavioral interfaces after the drill operation is listed twice, one time for the execution trace that does not involve drilling (block numbers indicated with suffix "a") and one time shifted by one step in case the drilling operation is performed (suffix "b"). The representation of the control program makes it harder to read than the functionally equivalent program from Algorithm 4.2.

Another limitation of the current approach is that the number of ECUs directly influences the number of predicates and hence the number of states. This means that synthesis time is exponential in the number of ECUs in distributed synthesis. Although not considered in this work, further optimizations to the encoding of state variables could partially mitigate this problem.

Finally, the game-based synthesis algorithm being used generates a "Java-like controller program" [Che12]. The translation to a program to run on a *programmable logic controller (PLC)* is possible but not straightforward. A promising approach is to translate the control program to structured text (compare Section 2.1.4). In this work, we do not support native execution of synthesized control programs on PLCs. Instead, we remotely control PLCs from PCs as described later in Section 6.3.1.

Algorithm 5.7: Simplified centralized control program for the task specification $\mathfrak{T}_2$ "drill undrilled work pieces" (compare Section 4.4.3.2).

```
1  lever-push (x_wp1, π_L1.from, π_L1.to);                          // block 1
2  plate-rotate (x_wp1, π_RP1.p1, π_RP1.p2);                        // block 2
3  probe-height (x_wp1, π_H1.probe);                                // block 3
4  plate-rotate (x_wp1, π_RP1.p2, π_RP1.p3);                        // block 4
5  if height (x_wp1, x_small) then
6  |  drill (x_wp1, π_D1.drill);                                    // block 5a
7  end
8  if ¬height (x_wp1, x_small) then
9  |  plate-rotate (x_wp1, π_RP1.p3, π_RP1.p4);                     // block 5b
10 end
11 if height (x_wp1, x_small) then
12 |  plate-rotate (x_wp1, π_RP1.p3, π_RP1.p4);                     // block 6a
13 end
14 if ¬height (x_wp1, x_small) then
15 |  lever-push (x_wp1, π_L2.from, π_L2.to);                       // block 6b
16 end
17 if height (x_wp1, x_small) then
18 |  lever-push (x_wp1, π_L2.from, π_L2.to);                       // block 7a
19 end
   /* nothing to do */                                              // block 7b
```

5.8. Summary

This chapter answered the question how (decentralized) control programs are automatically derived from a formal description of a MAL and the production task to achieve. For this purpose, we apply a game-based synthesis approach with the solver GAVS+. Synthesis is performed in four steps, namely checking of model constraints, translation of model to solver input language, game-based solving and translation of synthesis result to source code for the respective control programs. If a specification is feasible, the solver generates an FSM that implements the specification, otherwise it reports the specification as infeasible. The presented approach allows synthesizing centralized and decentralized control programs. Furthermore, guarantees on the performance of the synthesized control programs, for example with respect to worst-case execution time or worst-case power consumption, are provided using quantitative synthesis. Chapter 7 presents evaluation results for these scenarios.

The source code that is generated by the presented workflow contains references to state space variables and the triggering of actuations in the plant. The next chapter fills the remaining gap between the generated source code and an executable program by mapping the source code to the respective platform. For his purpose, code is automatically generated that implements the state space variables and actuations.

5.9. Related Work

The ideas of **game theory** were originally developed in the 1920s and found a broad audience when being successfully applied in modeling of decision situations during the Second World War. Since then, many distinct research areas in the context of game theory have formed, from analysis of people's decisions over evolution of genes to computer science [Tho84]. Game theory has earned a lot of attention in computer science in the past years. One of the reasons for this is that computational power of modern computers allows synthesizing strategies for games much faster than it was possible in the past. Using a suitable abstraction, this allows such approaches to be applied to real-world problems. This is illustrated by the fact that a number of tool implementations from the field of game theory and verification are presented every year at well-known computer science conferences such as CAV [MS12, SV13] and SPIN [BR13].

As summarized in [JGWB07], early works on **controller synthesis** include the one from Büchi and Landweber [BL67] as well as Rabin [Rab69]. Twenty years later, Pnueli and Rosner applied it to discrete event systems [PR89] and used specifications in *linear temporal logic (LTL)* to synthesize a controller that reaches a certain goal independent of the actions of the (adverse) environment. Shortly afterwards, they extended the approach to distributed systems [PR90]. Apart from distributed games (e.g., Mohalik and Walukiewicz [MW03]), controller synthesis has also been applied to timed systems (e.g., Asarin, Pnueli and Sifakis [AMPS98]).

In 1992, Rosner proved that synthesis of LTL formulas is 2EXPTIME-complete [Ros92]. In order to reduce the runtime complexity of LTL synthesis, subsets of LTL were analyzed in greater detail. Piterman, Pnueli and Sa'ar's synthesis of reactive(1) designs [PPS06] is a recent example that is based on a symbolic algorithm for synthesis on a subset of LTL of cubic complexity in the size of the state space [JGWB07].

Recent LTL-based modeling and synthesis tools include Anzu [JGWB07] that uses the approach from [PPS06], Acaia+ [BBF$^+$12] that reduces LTL reachability and synthesis problems to safety games and GAVS+ [CKLB11] that solves various types of games and allows to synthesize suitable strategies.

While most of these approaches are motivated by problems from the real world, many tool implementations lack the extent of the approach presented in this work: starting from modeling of automation systems and tasks over synthesis up to the mapping to the actual target platform. However, common to all the implementations that target real-world applications is the reluctance of decision makers to apply novel technologies in areas where flawless operation is essential for economic success, especially if those technologies apply a completely different workflow. It is expected that this reluctance can only be overcome by stable, reliable and conservative approaches that offer immediate benefits and that allow incremental transition from the existing to the new technology. By basing upon existing technologies and standards, the approach presented in this work attempts to ease this transition and to show immediate benefits for reconfigurable automation systems.

CHAPTER 6

Platform Mapping and Execution

Contents

6.1. Platform Mapping Overview **106**
6.2. Generation of Platform Mapping Code **106**
6.3. Mapping of Behavioral Primitives **107**
6.4. Manually Written Platform Library **111**
6.5. Discussion and Application to Running Example **113**
6.6. Summary **114**

Overview

This chapter answers the following research question:

> Given a formal description of the structure, capabilities, *electronic control units (ECUs)* and communication infrastructure of a *modular assembly line (MAL)* as well as control primitives for each capability, how can abstract (possibly decentralized) control programs be automatically mapped to the ECUs of the MAL?

This question is motivated by the fact that the "code" output by the workflow described in Chapter 5 is not directly executable. It is rather a sequential representation of a finite state machine in which transition conditions are evaluated in context of state space variables and states trigger actions via function calls (compare Algorithms 5.1 and 5.2). What is still missing is a software layer that implements the variables and functions such that the variables represent the state of the real plant and the functions trigger actions in the real plant. This layer can be automatically generated from the models defined in Chapter 4.

6.1. Platform Mapping Overview

Algorithms 5.1 and 5.2 on page 76 show simplified versions of decentralized control programs synthesized using the approach in this work. Both algorithms consist of a sequence of function calls, where a function call has one of the following semantics:

- Invocations of *behavioral interfaces* trigger actions in the plant. Such actions are either *actuations* (e.g., lever-push-e1, plate-rotate-e1) or *sensor triggerings* (e.g., probe-height-e2).
- Barriers synchronize all decentralized control programs by using blocking wait. Notice that in centralized control programs, the barriers are effectively no-operations, because no synchronization is required.
- Send operations transmit information from one control program to another (e.g., send-height-to-e1). Receive operations are set up accordingly. Notice that send/receive operations are not required in centralized control programs, because the system state (as specified in the model) is fully observable.

The next section depicts how the implementation for these functions is realized.

6.2. Generation of Platform Mapping Code

When game-based synthesis is successful, we apply model-to-text transformation based on the process model (i.e., code generation) to automatically generate a C++ source file that contains the following functionality:

- **Main entry point:** In order to run the generated strategy, a suitable software program needs a main entry point (in C/C++ programs typically indicated by a function called main()). The main entry point is generated automatically and performs the following tasks:
  - Parsing of command line arguments. Command line arguments to the control program executable are used to determine which ECU in a decentralized execution should be run.
  - Initialization of the plant's state according to the initial conditions $c_i \in \mathcal{C}_\mathcal{I}$.
  - Initialization of the communication interfaces between the machine running the control program and the plant (if required).
  - Initialization of all module instances with the respective channel bindings (explained in detail below).
  - Establishment of network connections to all other ECUs if running in decentralized execution mode.
  - Execution of the synthesized control program.
  - Cleanup and exit.
- **Numeric object identifiers:** In order to uniquely refer to every object instance $x \in \mathcal{X}$ in the model, a numeric constant is generated for each of them. This includes operating positions, work pieces, devices and behavioral interfaces.

- **State variables:** The generated source file contains variable definitions for every predicate $v \in \mathcal{V}$. Variables are generated according to their arity:
  - For predicates with zero parameters, a simple Boolean variable is generated.
  - For predicates with one parameter, a map (e.g., bitmap, C++ std :: map) from the type of the first parameter to a Boolean value is created. If the predicate is binary encoded (compare Section 5.5.3.2), an integer variable whose value is one of the object constants as defined above may be created instead.
  - For predicates with two parameters, a map between the type of the first and the second parameter is generated. The presence of an item in the map indicates that the predicate value is true for this valuation, otherwise it is false. If the predicate is binary encoded, a separate map between the type of the second parameter and a Boolean value may be generated instead for every valuation of the first parameter.

  This approach minimizes the memory requirements for the state variables according to the analyses performed during game-based synthesis. Notice that predicates that have been determined as read-only are completely omitted, because the solver has already replaced them by constant values during game-based synthesis (compare Section 5.5.3.1). All predicate values are considered false by default to match the semantics of the game-based solver.

  In order to ensure consistent access to the state variables, a lock is generated for every variable data structure. This is of special importance with respect to parallel action execution (compare Section 6.3.2). A pessimistic locking strategy is currently implemented that protects every read and write access to a variable by a critical section.
- **Behavioral interface functions:** For every behavioral interface b, a C++ function is generated that implements the effects of b. More details are provided in Section 6.3.
- **Send and receive functions:** For every predicate, a pair of send and receive functions is generated that allows sending a specific valuation of a predicate to another ECU. These functions are invoked by the transfer actions introduced during model-to-model transformation as described in Section 5.3.3.

  Since the execution of a send operation depends on the state of the variables on the sending side which might not have the same value on the receiving side, the receiver typically does not know that a state transfer is about to happen. Hence, the listening for transfer operations is implemented as part of the inter-ECU synchronization process as illustrated later in Section 6.4.

6.3. Mapping of Behavioral Primitives

In order to execute a control program, each *behavioral primitive* $\mathsf{b} \in \mathcal{B}$ of every module type in the hardware model is mapped to an automatically generated C/C++ function $\mathsf{b}(\mathsf{A})$, where A is the lists of parameters for b. The function $\mathsf{b}_1(\mathsf{A})$ in lines 22ff of Listing 6.1 illustrates the individual steps that are performed. First, it invokes the function

b_1_impl() (line 24). The rationale for this design is explained later. b_1_impl() first runs the *primitive control functions (PCFs)* $\hat{p}_{\mathsf{b}_1}$ of b_1 (line 11). The structure of the PCF is discussed in detail below. After that, the effects $\mathcal{E}_{\mathsf{b}_1}$ and conditional effects $\widehat{\mathcal{E}}_{\mathsf{b}_1}$ of b_1 are applied (lines 12f). Finally, b_1_impl() returns the execution context to b_1(), which calculates the new total cost based on the sequential composition operator $\odot$ (line 28). Notice that since the parallelCost vector contains just a single element, the choice of the parallel composition operator $\otimes$ has no effect.

6.3.1. Primitive Control Functions

A primitive control function, denoted by $\hat{p}_{\mathsf{b}}$, is associated to every behavioral interface b (compare Definition D6 on page 49). It is responsible for causing the actual effects of b in the real plant, possibly returning a sensor input if the behavioral interface is a sensor triggering.

For the running example, the implementation of each PCF has already been informally described in Section 4.4. For example, the height sensor module's PCF was indicated as follows (compare Section 4.4.1.4):

1. Set digital output o_{setRod} to high.
2. Wait p_{holdTime} milliseconds (interpreting the value of the parameter as a number).
3. Sample the value of digital input $\iota_{\mathsf{isRodDown}}$.
4. Set digital output o_{setRod} to low.
5. If the sampled $\iota_{\mathsf{isRodDown}}$ signal was high, return sensor result value 1, otherwise return sensor result value 0.

Implementing the PCF is a manual task. It requires the domain knowledge of experts in mechatronics as indicated in Figure 3.1 (a) on page 32. For implementing the PCF, established software tools may be used as required, for example tools supporting IEC 61131-3 programming languages.

A PCF is very similar to a hardware driver in an *operating system (OS)*: it is a low-level piece of software that makes the functionality of a device accessible to high-level software components. For this purpose, it provides a clearly defined programming interface. PCFs provide the following programming interface:

- **I/O channel binding**: The caller of a PCF configures all its input and output channels. For example, the height sensor module's PCF allows to configure which physical *input/output (I/O)* interfaces are to be used for o_{setRod} and $\iota_{\mathsf{isRodDown}}$. This information is automatically extracted from the hardware model.

  Since a plant may contain multiple instances of the same module type, multiple configurations of the channel bindings may be present. For example, if a plant contains two height sensor modules, then we need to distinguish which one is to be triggered when the control program invokes the corresponding PCF. For this purpose, every module instance is assigned a unique device identifier that is used as an additional parameter in every behavioral interface (compare Section 5.3.2).
- **Triggering:** Each PCF offers a function that is called to trigger its execution. If

Listing 6.1: Functions generated to implement behavioral interface execution.

```
unsigned int totalCost = 0U;
std::vector<unsigned int> parallelCost;
Lock parallelCostLock;

unsigned int calculateCost(unsigned int currentCost,
        const std::vector<unsigned int>& parallelCost) {
        return ⊙(currentCost, ⊗(parallelCost));
}

void b1_impl(A1) {
        p̂b1(A1);
        applyEffects(Eb1, A1);
        applyConditionalEffects(Êb1, A1);

        // Record cost in parallelCost using a critical section
        {
                ScopedLock lock(parallelCostLock);
                parallelCost.push_back(ηb1);
        }
}

void b1(A1) {
        parallelCost.clear();
        b1_impl(A1);

        // Calculate new total cost according to ⊙
        // (parallelCost contains one element, hence ⊗ is irrelevant)
        totalCost = calculateCost(totalCost, parallelCost);
}

void b2_impl(A2) {
        ...
}

void b2(A2) {
        b2_impl(A2);
        totalCost = calculateCost(totalCost, parallelCost);
}

void parallel_b1_b2(A1, A2) {
        parallelCost.clear();

        TaskGroup tg;
        tg.createTask(&b1_impl, A1);
        tg.createTask(&b2_impl, A2);
        tg.resumeAll();
        tg.wait();

        // Calculate new total cost according to ⊙ and ⊗
        totalCost = calculateCost(totalCost, parallelCost);
}
```

required, additional parameters from the behavioral interface are passed to the PCF when it is executed. In the tool accompanying this work, these parameters are called *hardware parameters* and are specified in a module's operating positions.

For example, the PCF for a storage rack with a number of physical operating positions that is managed by a robot may be implemented in a generic way and require a parameter specifying which operating position to navigate to. In this case, it is meaningful to treat the concrete position to navigate to as a parameter that is derived from the respective positional parameter of the triggering behavioral interface. In the software tool accompanying this work, this feature is implemented as an additional property of operating positions.

Depending on the type of the module being controlled and the type of ECU, different realizations of a PCF are possible. If the ECU is an (industrial) *personal computer (PC)*, the PCF is typically realized as a local control loop in a different thread or process that interacts with the environment using special I/O or communication adapters. If the ECU is a microcontroller, the PCF is typically a simple C/C++ function that manipulates the control unit's digital and/or analog I/O channels. If the ECU is a *programmable logic controller (PLC)*, two scenarios are possible: either the control program runs directly on the PLC and triggers function blocks as appropriate or it runs on a separate device such as a PC and remotely controls the PLC, for example via *open platform communications (OPC)* [IL01] or Siemens *Multi-Point Interface (MPI)* using third-party software libraries such as *libnodave* [Her13]. In the latter case, the remote control program changes the values of PLC-internal variables, which are periodically checked by the program running on the PLC. The PLC then executes the commands and stores the results in other variables that are subsequently queried by the remote control program. Some PLCs (e.g., Siemens SIMATIC) even allow directly manipulating the output channels and read the input channels of the PLC. I this case, no PLC program or PLC programming experience is required.

6.3.2. Parallel Execution

If a specification with parallel execution capabilities is selected (i.e., $\sigma \in \{\sigma_{\mathrm{rp}}, \sigma_{\mathrm{rcp}}\}$, compare Section 4.3.2), then the platform mapping needs to ensure that the respective behavioral interfaces are actually executed in parallel. For this purpose, the underlying platform needs to support explicit (quasi-)parallelism, for example in form of multithreading (e.g., on a PC) or function blocks (e.g., on a PLC with IEC 61131-3 support).

For every set of behavioral interfaces to execute in parallel, we generate a function that invokes the individual behavioral interfaces as parallel tasks, blocks until the individual tasks have finished and calculate the new total cost. The function $\mathrm{parallel\_b_1\_b_2}()$ in line 40 of Listing 6.1 illustrates the parallel execution of two behavioral interfaces b_1 and b_2. $\mathsf{A}_i = (\mathsf{a}_{\mathsf{b}_i 0}, \mathsf{a}_{\mathsf{b}_i 1}, \ldots, \mathsf{a}_{\mathsf{b}_i a-1})$ is the list of parameters for b_i. The TaskGroup class (line 43) provides functionality to launch multiple tasks in parallel and wait for their completion. Newly created tasks are in suspended state by default. The Lock class (line 3) is non-copyable and can be owned by at most one task at a time. The

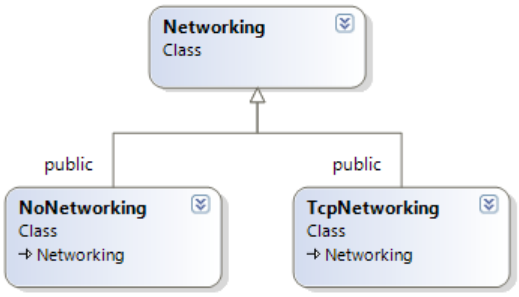


Figure 6.1.: Inter-ECU networking class hierarchy.

ScopedLock class (line 17) acquires the given lock on construction, blocking as long as it is not available. The lock is automatically released at the end of the current scope.

6.4. Manually Written Platform Library

In addition to the automatically generated content listed above, the platform mapping consists of a manually written software library for the following aspects:

- **Communication between decentralized control programs:** Inter-ECU communication in decentralized execution mode is twofold: on the one hand, the execution of all ECUs needs to be synchronized at the granularity of the blocks generated by the game-based solver (compare Algorithm 5.7 on page 102). On the other hand, information about the state of an ECU needs to be communicated to another ECU. Since the execution of a transfer operation depends on the state of the variables on the sending side, the receiver typically does not know that a state transfer is about to happen. Hence, we distinguish between two types of messages.

  When an ECU waits for synchronization with other ECUs (i.e., it executes a `barrier()` statement), it either receives a synchronization message, in which case no transfer from the respective ECU happens in the current step, or it receives one or multiple transfer message before the synchronization message, in which case it updates its local state accordingly. This approach allows communicating the necessary information from the sender to the receiver before the receiver attempts to evaluate the guard(s) of the next execution block.

  Depending on the type of replication strategy used (compare Section 5.3.3), either the complete state is transferred (token-based replication of the full state) or individual predicates (token-based replication at granularity of predicates) or predicate values are exchanged (pessimistic replication analysis).

  Figure 6.1 shows the C++ class hierarchy for this implementation. The `Networking` class is initialized with the set of ECUs to communicate with and introduces purely virtual functions for connecting to and disconnecting from all other ECUs, sending and receiving data between ECUs as well as waiting for the synchronization barrier. The `NoNetworking` class is used in case a centralized control program

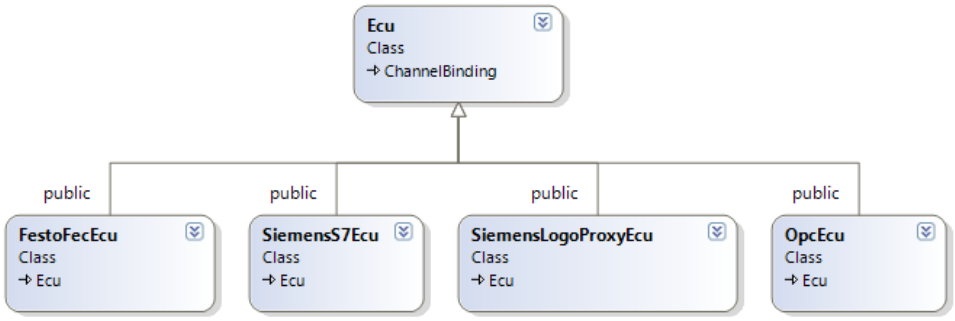


Figure 6.2.: ECU class hierarchy.

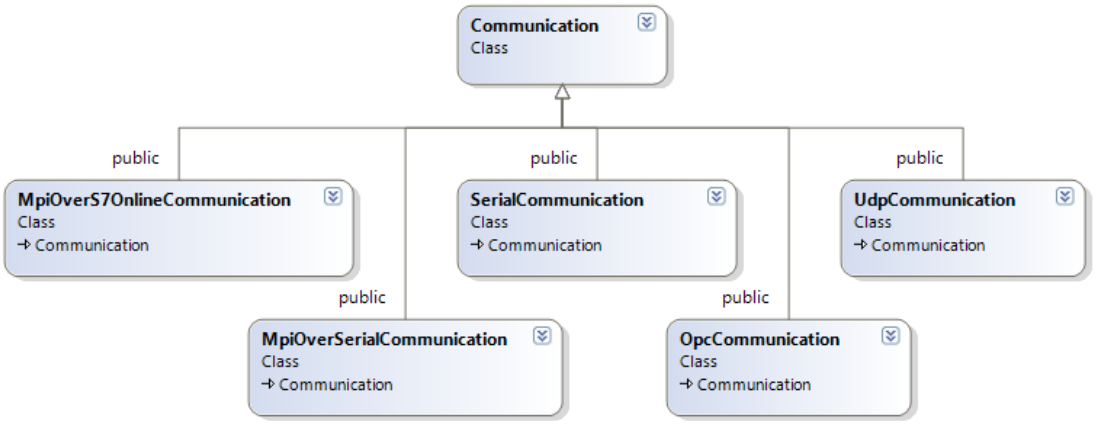


Figure 6.3.: Communication class hierarchy.

is generated; in this case all functions are implemented as no-operations. The TcpNetworking class implements connection-oriented sockets to all other ECUs, hence forming a fully connected graph between the ECUs. Connection oriented communication is preferred over connectionless communication, because inter-ECU synchronization requires reliable transfer.

- **Communication between control program and ECU:** If an ECU is remotely controlled as indicated in Figure 5.1 (b) on page 75, the control program(s) need(s) to be able to communicate with the ECU(s). The implementation currently supports remote control of ECUs of the families *Siemens S7-300* (using MPI, *libnodave* [Her13] and OPC [IL01]), *Festo FC 640* (via UDP/IP; TCP/IP not supported by the PLC) and *Siemens LOGO!* (using digital I/O via a custom "Siemens LOGO! Proxy"). Figure 6.2 shows a class diagram of the implementation. The Ecu class introduces a number of purely virtual functions for marshaling and demarshaling of data being sent to or received from an ECU. The subclasses of Ecu in turn rely on classes derived from Communication (compare Figure 6.3) to implement the actual communication. This design allows easily combining arbitrary communication mechanisms with arbitrary ECUs. The subclasses of Communication

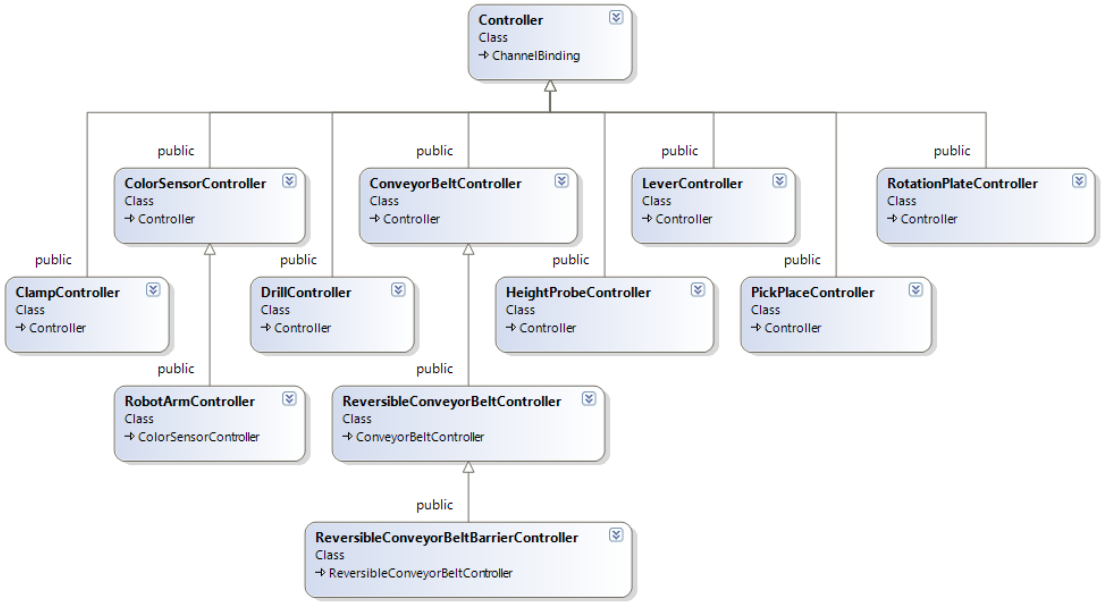


Figure 6.4.: Primitive control function class hierarchy.

are initialized with the address of the ECU to control an subsequently establish the respective communication channels. Furthermore, we provide a C++ class Controller that forms the base for the implementation of primitive control functions. The next section describes this class in detail.

- **Utility implementation:** An implementation for the C++ classes Lock, ScopedLock and TaskGroup needs to be provided as a base for parallel execution of behavioral interfaces.

6.5. Discussion and Application to Running Example

Figure 6.4 shows a *Unified Modeling Language (UML)* class diagram of all PCFs implemented in order to realize the platform mapping for the automation components in the running example and the other plants evaluated along with this work. The Controller class provides the basic interface for all PCFs. It derives from a class ChannelBinding that implements the channel bindings as described in Section 6.3.1. The subclasses implement the respective control functions for the individual hardware modules in the plant. Whenever an action is executed, the respective Controller object receives a reference to an object of class ECU and is hence able to read or write the respective I/O channels on the target ECU. See the actual implementation accompanying the MGSyn tool for details.

6.6. Summary

This chapter answers the question how a platform abstraction layer for execution of synthesized control programs for MALs is semi-automatically generated from a formal model of a MAL. Support PCs, PLCs and microcontrollers as target platforms has been illustrated.

Although the largest part of the platform mapping layer can be automatically derived from the MAL model, some parts – namely the low-level driver layer – need to be manually written once per hardware module. This is by design, because the realization of actions on a specific platform is typically quite different from another platform. Consider for example the driver layer of a microcontroller versus a PLC:

- On a microcontroller, the driving of a sensor or actuator is realized via digital and analog I/O interfaces with hardware interrupt support. Implementing a control algorithm typically implies a looped execution of a function in the program. Hence, it is most efficient to implement the driver layer directly in hardware-near C/C++ code and offer respective communication interfaces (e.g., Ethernet, RS232) for triggering them.
- On a PLC, the driving of a sensor or actuator is typically realized by a function block in an IEC 61131-3 language [IEC03a]. Implementation of a control algorithm is realized by higher-level function blocks. Hence, it is most efficient to implement the driver layer in an IEC 61131-3 language and offer respective *application programming interfaces (APIs)* (e.g., OPC) for triggering them.

Introducing a model that covers these aspects is feasible, but would most probably make specification overly complex. For this purpose, we decided to leave this part to a software developer in order to retain the possibility to use appropriate tooling and debugging methods.

Notice that the generated mapping layer is independent of the task to execute. Hence, it only needs to be regenerated when either the hardware model or the plant model changes. This approach fosters modularity and reuse of code.

CHAPTER 7

Realization and Evaluation

Contents

7.1. Model-driven Development Tool **MGSyn** 116
7.2. Simulation of Control Program Execution 118
7.3. Evaluation Overview 119
7.4. Evaluation of the Running Example 121
7.5. Evaluation of Circular Material Flow Example: Focus on Parallel Execution 122
7.6. Evaluation of Bidirectional Material Flow Example: Focus on Decentralized Execution 126
7.7. Summary 132

Overview

This chapter depicts how the concepts from the previous chapters are implemented in a model-driven development tool called MGSyn. MGSyn serves as a end user front-end for game-based synthesis and covers the whole workflow from model specification over synthesis to simulation and execution. This chapter also provides some technical details on the realization of the platform mapping on the respective target systems, namely *personal computers (PCs)*, microcontrollers and *programmable logic controllers (PLCs)*.

The second part of this chapter is dedicated to in-depth evaluations. It compares sequential and parallel action execution, quantitative and non-quantitative synthesis, centralized and decentralized control program synthesis, simulation and execution on real hardware as well as remote controlled and local execution.

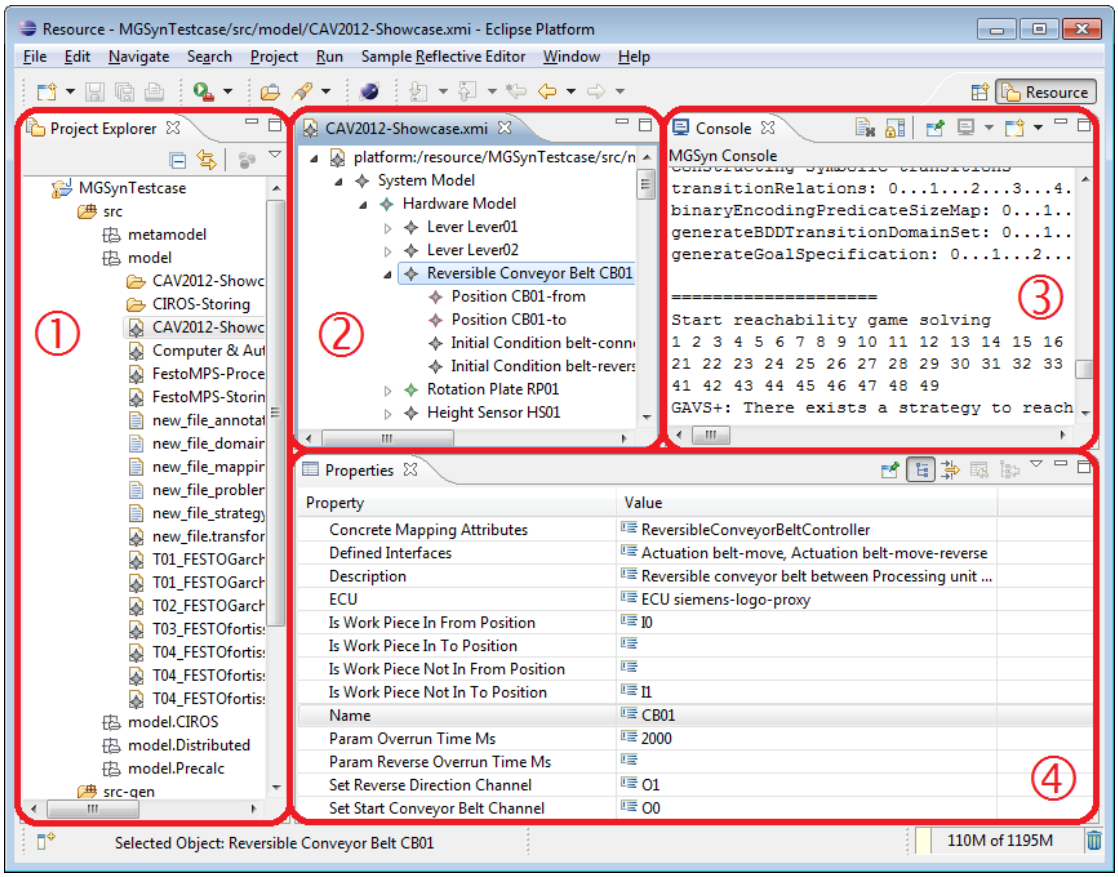


Figure 7.1.: MGSyn main application window: ① *Project Explorer* with models and generated files, ② model editor with expanded and collapsed model elements, ③ *Console* with synthesis progress and result, ④ *Properties* window showing the properties of model element "Reversible Conveyor Belt CB01".

7.1. Model-driven Development Tool MGSyn

Along with this work, the model-driven development tool MGSyn, for which development was originally started by Chih-Hong Cheng in 2011 [Che12], was heavily extended according to the approach presented in this work [CGR+12a]. MGSyn is the abbreviation for "Model, Game and Synthesis" [for13]. The purpose of this tool is to capture the whole development workflow from specification of the hardware, plant and task model over game-based synthesis to platform mapping of the resulting control strategy. Implementing a model-driven tool is a very important step to make the theoretic results illustrated in this thesis accessible to control engineers.

MGSyn is implemented as a plug-in to the Eclipse Platform and is based on *Eclipse Modeling Framework (EMF)* [BSM+04]. Parts of it are licensed under the Eclipse Public License. All other parts are licensed under the GNU General Public License, Version 3.0. The tool is available for download at `http://mgsyn.fortiss.org/`. Figure 7.1 shows a screenshot of the tool.

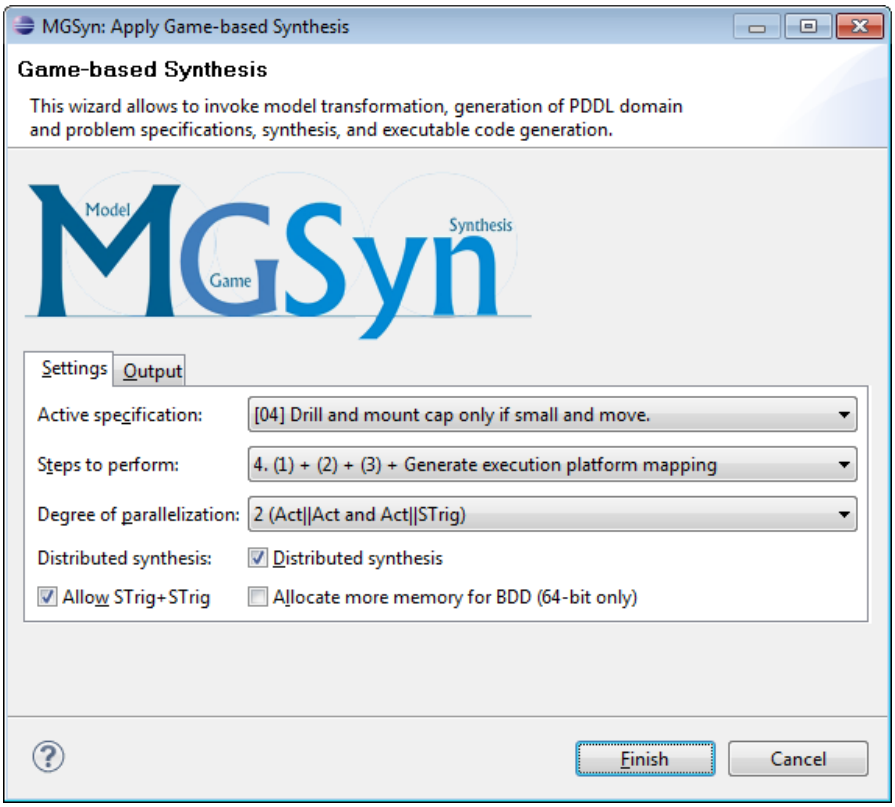


Figure 7.2.: MGSyn game-based synthesis wizard dialog.

In order to invoke game-based synthesis on a model file (represented in *XML Metadata Interchange (XMI)* format), the command *Verfication → MGSyn: Apply Game-based Synthesis* is chosen from the popup menu in the *Project Explorer*. Subsequently, the configuration dialog as illustrated in Figure 7.2 pops up and allows the user to adjust options for game-based synthesis:

- **Active specification:** This drop-down box allows the user to select one of the problem specifications defined in the model. This allows to quickly switch between different control strategies. Problem specifications are either quantitative or non-quantitative. For quantitative specifications, the cost bound $\eta_{\max}$ is specified in the model.
- **Steps to perform:** This drop-down box allows to select which steps related to game-based synthesis are to be performed:
  1. **Transform model** applies constraint checking and model transformation as specified in Sections 5.2 and 5.3 and outputs the transformed model as an XMI file (named `new_file.transformed.xmi` by default).
  2. **Generate synthesis model (*Planning Domain Definition Language (PDDL)* domain and PDDL problem)** performs model transformation and translates the resulting model into a PDDL domain file and a PDDL problem specification as discussed in Section 5.4.

3. **Synthesize strategy in high-level control format** performs all the steps from above and runs game-based synthesis in order to obtain an abstract control program as specified in Section 5.5.
4. **Generate execution platform mapping** performs all the steps from above and generates the files required for execution platform mapping as discussed in Sections 5.6 and 6.1.

- **Degree of parallelization:** This drop-down box allows to select the degree of parallelization d. Currently values of 1 (no parallelization), 2 (parallelization of two behavioral interfaces) and 3 (parallelization of two behavioral interfaces plus parallelization of three actuations) are currently supported.
- **Distributed synthesis:** This check box specifies whether decentralized control programs should be generated as opposed to a centralized control program. The state of this check box influences the model transformation and post-processing performed as discussed in Section 5.3.
- **Allow STrig+STrig:** This check box specifies whether two sensor triggerings can be executed in parallel if $d \geq 2$. The reason why this is handled separately is that the parallel triggering of two sensor inputs requires specific measures in the PDDL specification and the platform mapping.
- **Allocate more memory for BDD (64-bit only):** This check box assigns more memory to the *binary decision diagram (BDD)* data structure used internally in the game solving engine *Game Arena Visualization and Synthesis Plus! (GAVS+)*. This option is only meaningful on 64-bit platforms with large address space. It allows solving more complex problems at the cost of higher memory consumption.

7.2. Simulation of Control Program Execution

MGSyn distinguishes between *execution mode* and *simulation mode*. In execution mode, the *primitive control functions (PCFs)* are executed as specified in the model. This means that actuations are visible in the real plant and sensor triggerings retrieve values from the real plant.

In simulation mode, the synthesized control programs execute directly on the development machine. In this case, the PCFs of all actuations are replaced by no-operations and the PCFs of all sensor triggerings are replaced by a prompt asking the user to select one of the possible sensor inputs. This means that whenever a sensor triggering is executed, the user may specify one of the valid values from the model in order to simulate a sensor input. It allows to verify the functionality of the synthesized control program by testing all possible execution traces. This is very useful to evaluate control programs for plants that do not actually exist or have not yet been built.

Figure 7.3 shows a screenshot of the simulated execution of four decentralized control programs for the specification "drill the object at position A in the storage rack and mount a cap if it is red and small" (corresponds to task $\mathfrak{T}_4$ introduced later in Section 7.6). The individual control programs are mapped to *operating system (OS)* processes in this case. Processes communicate with each other via *Transmission Control*

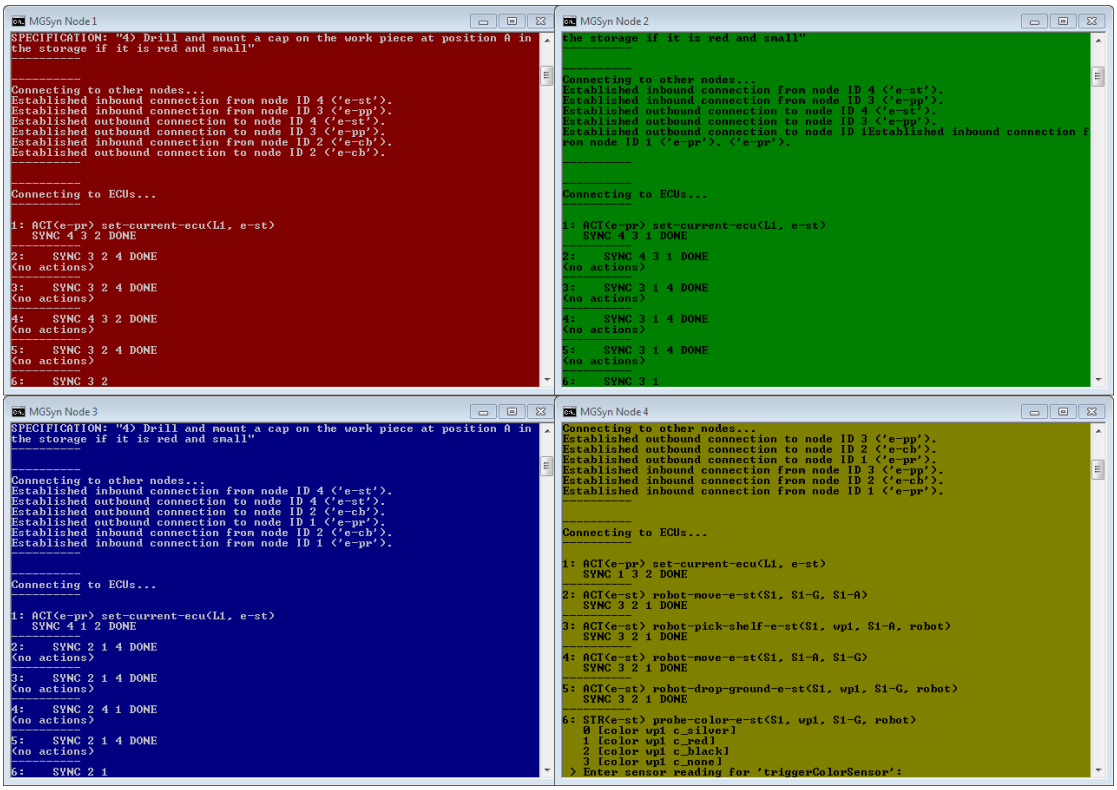


Figure 7.3.: Simulating execution of decentralized control programs for four ECUs: the control program for each ECU runs as a separate process. Inter-process communication is realized via TCP/IP.

Protocol (TCP)/Internet Protocol (IP) in this example. All control programs synchronize with each other during startup. This means that execution of the control programs starts as soon as all control programs are "online".

7.3. Evaluation Overview

7.3.1. Scope

In the following, some in-depth evaluations of the approach presented in this thesis are provided. Analyses are performed with respect to various synthesis and platform mapping parameters, such as:

- **Parallel vs. sequential action execution:** If a degree of parallelization of $d_{\mathfrak{T}} \geq 2$ is selected, multiple behavioral interfaces may execute in parallel. In setups that are suitable for parallelization, this reduces the average number of execution steps.
- **Quantitative vs. non-quantitative synthesis:** In quantitative synthesis, cost $\eta_{\mathfrak{b}_{\mathfrak{H}}}$ is associated to each behavioral interface $\mathfrak{b}_{\mathfrak{H}}$ and sequential and parallel composi-

tion operators $\odot_{\mathfrak{T}}$ and $\otimes_{\mathfrak{T}}$ as well as a cost bound $\eta_{\max\mathfrak{T}}$ is selected. Every successfully synthesized control program is guaranteed to not exceed the cost bound.

- **Centralized vs. decentralized control program synthesis:** A centralized control program is a single control program that "remotely controls" all *electronic control units (ECUs)* $\hat{u}_{\mathfrak{P}}$ in a plant. In decentralized control program synthesis, an individual control program is synthesized for every ECU as well as respective communication patterns for state exchange at runtime.
- **Simulation vs. execution on real hardware:** In simulation mode, synthesized control programs do not interact with a "real" plant; required sensor inputs are supplied by the user. If executed on real hardware, synthesized control programs trigger actuations in the real plant and read sensor values from the plant's environment.
- **Remote control vs. local execution:** If ECUs are remotely controlled, the control program is connected to every ECU via a communication network and exchanges control signals with the ECU. A proxy application is running on the ECU that handles the requests. This execution model is good for debugging and step-wise execution of the control programs. In local execution mode, the control program is directly executed on the ECU. For a comparison of different execution models, see also [GBWK09].

In case of feasibility, synthesis times presented in the following include execution of the complete workflow including C/C++ code generation for execution on real hardware or simulation. Worst case numbers of moves were directly extracted from the generated strategy. Worst case costs were derived by inspecting all possible paths in the generated strategy using simulation. The Java Virtual Machine is by default started with a parameter of *-Xmx3072m* (3.0 GB maximum Java heap size).

7.3.2. Demonstration Platform: Festo Modular Production System

The demonstration platforms presented in the following are not only of theoretical nature; most have been built up in form of real plants and execution of the control programs verified on the respective ECUs. The plants are built of components from the *Festo Modular Production System (MPS)* [Fes12]. MPS is designed as a learning environment for students and trainees in mechanical engineering, electrical engineering, control theory and mechatronics. As such, it is built from industrial-grade components which are used in similar form in real plants. Hence, the MPS provides a good compromise between flexibility and truth to the original.

As the name implies, the MPS is built from modular components that are combined to form so-called stations. Each station has a dedicated purpose and can be combined with a number of neighboring stations. The "goods" being processed are small circular work pieces that are transported through the plant using conveyor belts, levers, rotary plates and robot arms. Supported processing steps include distribution of work pieces from a stacking magazine, testing of certain properties of work pieces, processing of work pieces (e.g., drilling, mounting), storing of work pieces in a rack and sorting of work pieces for delivery to customers.

Control of the named production steps is by default performed using industrial PLCs, one in each station. In order to implement the demonstration setups, we extended the standard Festo MPS components and PLC programs in various ways with customized equipment and code. The respective modifications are introduced in the individual sections.

7.4. Evaluation of the Running Example

This section evaluates the tasks $\mathfrak{T}_1$ and $\mathfrak{T}_2$ from Section 4.4.3 on page 66.

Task 1 & 2: Drill Work Pieces and Drill Small Work Pieces

Scenarios:

The following scenarios are analyzed in this evaluation.

1a) $\mathfrak{T}_1$**, one work piece:** Synthesize a strategy according to goal g_1 of task $\mathfrak{T}_1$.

1b) $\mathfrak{T}_1$**, two work pieces:** Synthesize a strategy that requires two work pieces to be drilled instead of one in order to demonstrate parallel execution of behavioral interfaces. For this purpose, process model $\mathfrak{P}_1$ is extended to obtain $\mathfrak{P}_1'$ as follows:

- Introduce a behavioral interface plate-rotate-two to rotate two work pieces simultaneously.
- Introduce a behavioral interface output to "output" work pieces at the slide. This is necessary to free the operating position of the slide.

For details, please refer to the models shipping with MGSyn.

2a) $\mathfrak{T}_2$**, one work piece:** Synthesize a strategy according to goal g_2 of task $\mathfrak{T}_2$.

2b) $\mathfrak{T}_2$**, two work pieces:** Synthesize a strategy that requires two work pieces to be drilled instead of one if they are small. This evaluation uses the adapted process model $\mathfrak{P}_1'$ from scenario 1b).

Evaluation results:

Table 7.1 summarizes the results for tasks $\mathfrak{T}_1$ and $\mathfrak{T}_2$. The "WC moves" column specifies the worst-case number of moves (i.e., actions of *player* Controller) executed in order to reach the goal. The "Num. vars" column names the number of Boolean variables in the state space. Synthesis times generally vary by about $\pm$ 2 % due to effects such as caching, paging and multitasking. Hence, the presented times in this chapter are best ones out of three runs on a 3.2 GHz system with 8 GB of RAM unless noted otherwise.

If just one work piece is considered, the synthesized strategy is the same with and without parallelization; only synthesis time differs slightly. This is expected, because the model has no potential for parallel execution with just a single work piece. In the scenarios with two work pieces, synthesis time is much higher, because the number of variables is higher (24 as opposed to 13). This difference is due to the at, drilled and height predicates being represented for both work pieces and the occupied predicate not being

Table 7.1.: Synthesis results for tasks $\mathfrak{T}_1$ and $\mathfrak{T}_2$ in the running example. Times are best ones out of three runs on a 3.2 GHz system with 8 GB of RAM.

Experiment	σ	d	**WC moves**	**Num. vars**	**Synthesis time (sec)**
1a) One work piece, unconditional drill	σ_{r}	1	6	13	1.2
	σ_{rp}	2	6	13	1.5
1b) Two work pieces, unconditional drill	σ_{r}	1	11	24	15.1
	σ_{rp}	2	10	24	15.7
2a) One work piece, conditional drill	σ_{r}	1	7	13	1.4
	σ_{rp}	2	7	13	1.5
2b) Two work pieces, conditional drill	σ_{r}	1	13	24	15.4
	σ_{rp}	2	11	24	15.8

eligible for binary encoding anymore. Since synthesis time increases exponentially in the number of variables in general, the synthesis time increases by more than an order of magnitude. On further analysis, more than 90 % of that time is spent in the so-called construction of symbolic transitions in the solver, that is the phase in which the game transition graph is constructed. The actual game-based solving then only takes a fraction of the total time. Parallelization allows to reduce the worst case number of moves from 11 to 10 (9 %) for task $\mathfrak{T}_1$ and from 13 to 11 (15 %) for task $\mathfrak{T}_2$ when two work pieces are processed. In the following, more complex examples are evaluated that have more potential for parallelization.

7.5. Evaluation of Circular Material Flow Example: Focus on Parallel Execution

The first complex example being evaluated is depicted in Figure 7.4. The real plant built from Festo MPS components is depicted in Figure 7.5. It consists of the two Festo MPS stations "Processing" and "Storage" as well as conveyor belts that interconnect the output of each station with the input of the respective other station. Each station is equipped with a PLC of type *Festo FEC FC640-FST* and the conveyor belts are controlled using two PLCs of type *Siemens LOGO! 12/24RC*. The Festo FC PLCs are connected to a PC running the control program via Ethernet. Since the Siemens LOGO! ECUs do not offer a documented communication interface that would allow remotely controlling them, the commands to execute are triggered by respective signals on the unused digital and analog input ports using a microcontroller-based setup that we call "Siemens LOGO! Proxy". For this purpose, an ATmega168 microcontroller with serial interface and a suitable control program is used. The specific properties of the whole setup are listed in the following:

- Circular material flow in counter-clockwise direction. This allows multiple work pieces to be processed independently without "blocking" each other. Hence, this scenario is particularly useful to evaluate synthesis with parallel execution.

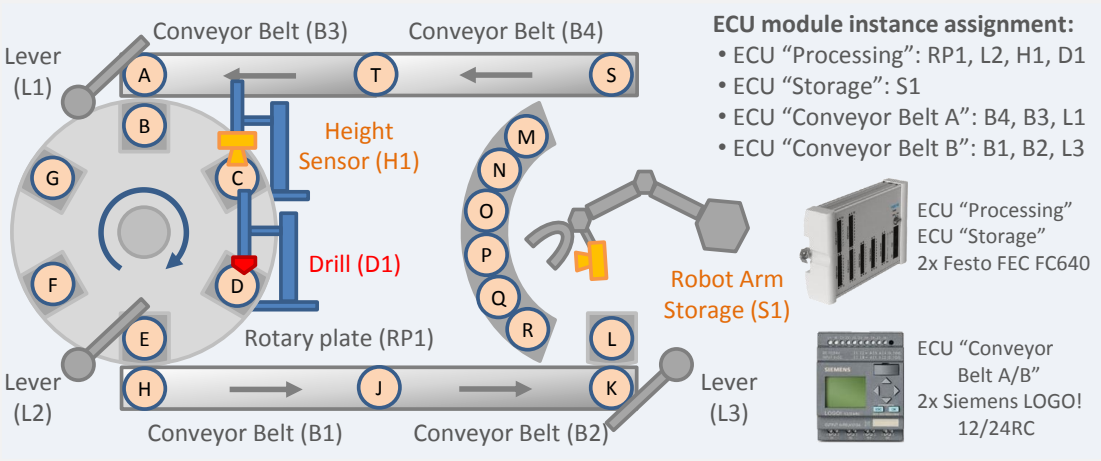


Figure 7.4.: Circular material flow example: in this plant, work pieces may move in a counter-clockwise way between two processing stations. The left side is almost identical to the running example from Section 3.4. The right side shows a storage station with six storage locations (operating positions M through R) and a robot arm that transports work pieces between the storage locations and the input (operating position L) and output locations (operating position S). In addition, a color sensor is mounted on the robot arm. The unidirectional conveyor belts (H $\rightarrow$ J, J $\rightarrow$ K, S $\rightarrow$ T and T $\rightarrow$ A) transport work pieces between the processing stations (ECU icons by [Ele14, Sie14b]).

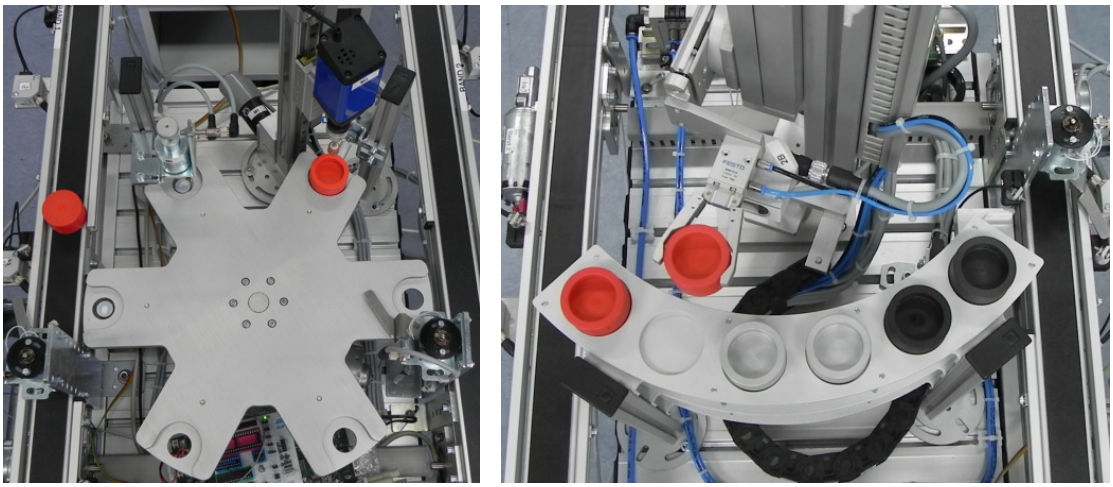


Figure 7.5.: Photos of real plant implementing circular material flow: on the left, Processing unit with levers, height sensor and drill. On the right, Storage unit with lever and robot arm. On the left, a small red work piece is located underneath the drill (operating position D in Figure 7.4) and a large red work piece is located on the conveyor belt (just arriving at operating position A). On the right, the storage contains five small work pieces of different colors (red, silver, black) and the robot arm carries a small red work piece.

- 11 hardware modules: 4 conveyor belts, 3 levers, rotary plate, height sensor, drill, robot arm storage.
- 3 types of sensor inputs: work piece height and color, storage rack occupancy.

In order to represent all hardware modules, the following new model elements are introduced in addition to the ones introduced in Section 4.4.1. For brevity, the formal modeling of those elements is omitted. Please refer to the accompanying tool MGSyn [for13] for details, which ships with the respective models.

- Hardware modules:
  - Robot arm storage: models the storage, the robot arm and the color sensor that is mounted at the robot. In the following evaluations, operating positions Q and R are omitted from the model, because they are not needed. This improves synthesis time, because the number of variables is smaller.
- Behavioral interfaces:
  - robot-move, robot-pick-ground, robot-drop-ground, robot-pick-rack, robot-drop-rack: move robot and pick up respectively drop work pieces. "Ground" means operating positions L or S and "rack" refers to positions M through R.
  - belt-move, belt-move-robot: move conveyor belts; distinguishing is necessary here, because the robot arm might block operating position S and hence belt-move-robot, which is used for B4, ensures through a precondition that the robot arm is not at that place.
  - plate-rotate-two: rotate the plate with two work pieces on it.
  - probe-color: detect the color of a work piece (only possible if work piece is in one of the ground positions).
  - probe-rack: use the color sensor to detect if a storage location is occupied.
- Predicates:
  - in-robot: stores the current position of the robot arm (binary encoded).
  - gripper-occupied: stores whether the gripper of the robot arm is occupied by a work piece. The gripper itself is modeled as an additional operating position of the robot arm storage.
  - carry: stores which work piece is carried by the robot arm (binary encoded).
  - belt-located: predicate that denotes that two operating positions are connected via a conveyor belt (read only).
  - color: stores the color of a work piece (binary encoded).
  - rack-occupied: stores the occupancy state of a rack in the storage.
- Costs:
  - robot-move, belt-move, belt-move-robot: cost 3.
  - plate-rotate, plate-rotate-two: cost 2.
  - All other behavioral interfaces (including sensor triggerings): cost 1.

  These costs roughly correspond to the execution time of the individual actions and/or their power consumption.

Task 3: Drill Small and Sort by Color with Output on Overflow

Informal task description:

Drill a work piece x_{wp1} initially located at operating position T if it is small and move it to operating position J. In addition, sort a work piece x_{wp2} initially located at operating position J by color into the storage rack (a red work piece to either operating position M or N, a black work piece to either operating position O or P). If the respective spaces are already occupied, move the work piece to operating position T instead. This scenario is an extended version of the one in [CGB13][1].

Formal task description:

$$\mathfrak{T}_3 = (\mathfrak{P}_0, \mathcal{C}_{\mathcal{I}3}, \sigma_3, g_3, \odot_3, \otimes_3, \eta_{\max 3}, d_3) \tag{7.1}$$

$$\mathcal{C}_{\mathcal{I}3} := \{\mathsf{at}(x_{\mathsf{wp1}}, \pi_{\mathsf{T}}), \mathsf{occupied}(\pi_{\mathsf{T}}), \mathsf{at}(x_{\mathsf{wp2}}, \pi_{\mathsf{J}}), \mathsf{occupied}(\pi_{\mathsf{J}})\} \tag{7.2}$$

$$\begin{aligned} g_3 := {} & (\mathsf{height}(x_{\mathsf{wp1}}, x_{\mathsf{small}}) \vee \mathsf{height}(x_{\mathsf{wp1}}, x_{\mathsf{large}})) \\ & \wedge (\mathsf{height}(x_{\mathsf{wp1}}, x_{\mathsf{small}}) \Leftrightarrow \mathsf{drilled}(x_{\mathsf{wp1}})) \wedge \mathsf{at}(x_{\mathsf{wp1}}, \pi_{\mathsf{J}}) \\ & \wedge (\mathsf{color}(x_{\mathsf{wp2}}, x_{\mathsf{red}}) \vee \mathsf{color}(x_{\mathsf{wp2}}, x_{\mathsf{black}}) \vee \mathsf{color}(x_{\mathsf{wp2}}, x_{\mathsf{none}})) \\ & \wedge (\mathsf{color}(x_{\mathsf{wp2}}, x_{\mathsf{red}}) \Rightarrow (\mathsf{at}(x_{\mathsf{wp2}}, \pi_{\mathsf{M}}) \vee \mathsf{at}(x_{\mathsf{wp2}}, \pi_{\mathsf{N}}) \\ & \quad \vee (\mathsf{rack\text{-}occupied}(\pi_{\mathsf{M}}, x_{\mathsf{yes}}) \wedge \mathsf{rack\text{-}occupied}(\pi_{\mathsf{N}}, x_{\mathsf{yes}}) \wedge \mathsf{at}(x_{\mathsf{wp2}}, \pi_{\mathsf{T}})))) \\ & \wedge (\mathsf{color}(x_{\mathsf{wp2}}, x_{\mathsf{black}}) \Rightarrow (\mathsf{at}(x_{\mathsf{wp2}}, \pi_{\mathsf{O}}) \vee \mathsf{at}(x_{\mathsf{wp2}}, \pi_{\mathsf{P}}) \\ & \quad \vee (\mathsf{rack\text{-}occupied}(\pi_{\mathsf{O}}, x_{\mathsf{yes}}) \wedge \mathsf{rack\text{-}occupied}(\pi_{\mathsf{P}}, x_{\mathsf{yes}}) \wedge \mathsf{at}(x_{\mathsf{wp2}}, \pi_{\mathsf{T}})))) \end{aligned} \tag{7.3}$$

Two remarks:

- The color x_{none} in equation (7.3) is added in order to model that fact that no work piece might be in front of the color sensor when it is triggered. Notice that the synthesized program will not be able to handle this situation, but in order to correctly model the sensor input, all possible "colors" need to be represented.
- Since the initial state for rack occupancy needs to be unknown (and not "unoccupied"), we model the occupancy state as a binary predicate $\mathsf{rack\text{-}occupied} : \Pi \times \mathcal{X}_{\mathsf{occupied}} \to \mathbb{B}$ with $\mathcal{X}_{\mathsf{occupied}} = \{x_{\mathsf{yes}}, x_{\mathsf{no}}\}$. Hence, the default value "all false" (i.e., neither $\mathsf{rack\text{-}occupied}(\pi, x_{\mathsf{yes}})$ nor $\mathsf{rack\text{-}occupied}(\pi, x_{\mathsf{no}})$) represents that the occupancy state is unknown for $\pi \in \Pi$.

Scenarios:

The following scenarios are analyzed in this evaluation. For semantics of cost, refer to Tables 4.4 and 4.5 on page 69.

[1] See also the following videos from execution of other task models on the same plant: `http://youtu.be/Sb3bre916o4` (storing by color), `http://youtu.be/Foenmw31rB4` (error handling), `http://youtu.be/daHLnx2IsIs` (processing and storing with occupancy check).

3a) **WCET optimization:** Synthesize a strategy that does not exceed a specified maximal *worst-case (WC) execution time (ET)* $\eta_{\max 3}$. The cost annotated to a behavioral interface corresponds to its ET. In this scenario, $\otimes_3 :=$ max is chosen as parallel composition operator, because parallel execution of multiple behavioral interfaces takes as long as the interface with the longest ET. $\odot_3 :=$ sum is chosen as sequential composition operator, because ET accumulates over sequential actions.

3b) **WC total power consumption optimization:** Synthesize a strategy not exceeding a given WC total power consumption $\eta_{\max 3}$. The cost annotated to a behavioral interface corresponds to its power consumption. $\otimes_3 :=$ sum is chosen as parallel composition operator, because parallel execution of multiple actions consumes the cumulative power of all executed actions. $\odot_3 :=$ sum is chosen as sequential composition operator, because power consumption accumulates over sequential actions.

3c) **WC peak power consumption optimization:** Synthesize a strategy not exceeding a given WC peak power consumption $\eta_{\max 3}$. The cost annotated to a behavioral interface corresponds to its power consumption. Again $\otimes_3 :=$ sum is chosen as parallel composition operator, because parallel execution of multiple actions consumes the cumulative power of all executed actions. $\odot_3 :=$ max is chosen as sequential composition operator, because we are interested in the worst-case power consumption at any point in time.

Evaluation results:

Table 7.2 summarizes the results for task $\mathfrak{T}_3$. The results show that up to 30 % (7 of 23 and 7 of 24) of the control moves can be parallelized and that parallelization requires typically more than twice the synthesis times for feasible specifications in these scenarios. Higher cost bounds require a slightly higher synthesis time, because more bits are required to represent the current cost during game-based synthesis. When the cost bound is very tight, the tool synthesizes a strategy with more, but cheaper moves (e.g., 17 instead of 16). The generated strategy for experiment 3c) significantly differs from the strategy for 3a) and 3b).

Due to its circular nature, this setup was specifically well suited for parallelization. One reason for this is that multiple work pieces can be simultaneously processed without the risk for one work piece to block another. In the following, an example with a linear architecture is evaluated. That example is less well suited for parallelization.

7.6. Evaluation of Bidirectional Material Flow Example: Focus on Decentralized Execution

The second complex example being evaluated is depicted in Figure 7.6. The real plant built from Festo MPS components is depicted in Figure 7.7. It consists of the three Festo MPS stations “Processing”, “Pick & Place” and “Storage” as well as a conveyor belt that interconnects the output of the processing station to the input of the Pick & Place

Table 7.2.: Synthesis results for task $\mathfrak{T}_3$ in the circular example. Results for experiments without parallelization and without cost model are provided for comparison. Times are best ones out of three runs on a 3.2 GHz system with 8 GB of RAM.

Experiment	σ_3	d_3	$\eta_{\max 3}$	$\odot_3$	$\otimes_3$	**WC moves**	**Num. vars**	**Synthesis time (sec)**
3a) WCET optimization	σ_{rcp}	2	32	sum	max	inf.[1]	55	22.0
	σ_{rcp}	2	33	sum	max	17	55	22.5
	σ_{rcp}	2	34	sum	max	16	55	22.1
	σ_{rcp}	2	35[2]	sum	max	16	55	22.2
3b) WC total power optimization	σ_{rcp}	2	45	sum	sum	inf.[1]	55	38.6
	σ_{rcp}	2	46	sum	sum	17	55	29.3
	σ_{rcp}	2	47	sum	sum	16	55	28.0
	σ_{rcp}	2	48	sum	sum	16	55	28.4
3c) WC peak power optimization	σ_{rcp}	2	2	max	sum	inf.[1]	46	14.9[3]
	σ_{rcp}	2	3	max	sum	20	49	16.3
	σ_{rcp}	2	4	max	sum	17	49	18.7
	σ_{rcp}	2	5[2]	max	sum	17	49	19.0
3a)/3b) without parallelization[4]	σ_{rc}	1	45	sum	N/A	inf.[1]	55	13.8
	σ_{rc}	1	46	sum	N/A	24	55	13.8
	σ_{rc}	1	47[2]	sum	N/A	23	55	13.8
3c) without parallelization	σ_{rc}	1	2	max	N/A	inf.[1]	46	6.8
	σ_{rc}	1	3[2]	max	N/A	23	49	8.3
Non-quantitative (without cost model)	σ_{r}	1	∞	N/A	N/A	23	49	8.3
	σ_{r}	2	∞	N/A	N/A	16	49	18.8

[1] Infeasible (i.e., no solution) due to the cost bound being too restrictive.
[2] The same strategy is generated for higher cost bounds, only synthesis time differs.
[3] Since it is not clear whether behavioral interfaces with cost 3 are used in the generated strategy, infeasibility cannot be directly decided from the cost bound.
[4] If parallelization is disabled, 3a) and 3b) collapse to the same scenario.

station. Each station is equipped with a PLC of type *Siemens S7-313C* as well as a microcontroller of type ARM Cortex M3 of type *STM32F107VCT6* on an *Olimex STM32-P107* board. The conveyor belt is steered by a *Siemens LOGO! 12/24RC* that is remotely controlled via unused input ports as in the previous example or by an ARM Cortex M3 microcontroller. In order for the microcontrollers to be able to interact with the plant, custom adapter boards and voltage converter boards to/from 3.3 V/24 V have been designed and built (compare Figure 7.7). Either the PLCs or the microcontrollers may be selected as controllers for the plant. The specific properties of this setup are:

- Bidirectional material flow: in order to transport work pieces forth and back, the conveyor belts have been equipped with additional electric components in order to switch the transport direction. In addition, the mechanical setup has been slightly modified to allow the robot arm of the storage station to push a work piece from operating position G to operating position H.

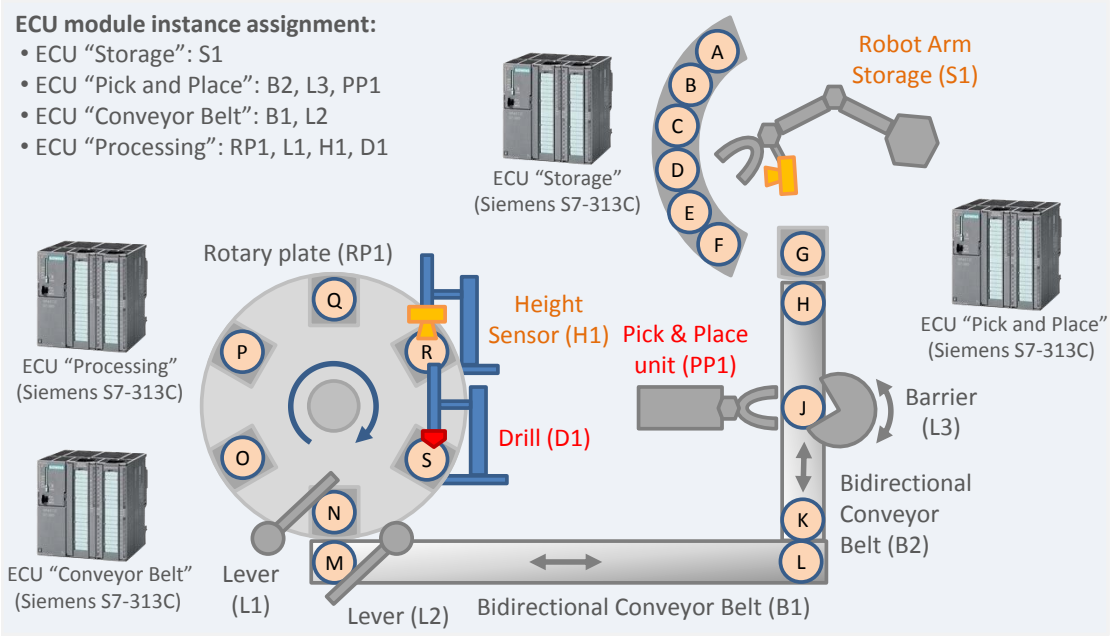


Figure 7.6.: Bidirectional material flow example: this plant has a linear material flow with a number of processing stations. It includes a robot arm-based storage (operating positions A to G), a pick and place unit that mounts caps on work pieces (at operating position J), two bidirectional conveyor belts (H ↔ K and L ↔ M) and a processing unit similar to the running example from Section 3.4 (N to S). In order to transport work pieces forth and back, the conveyor belts in this plant move forward and backward (icons by [Sie14a]).

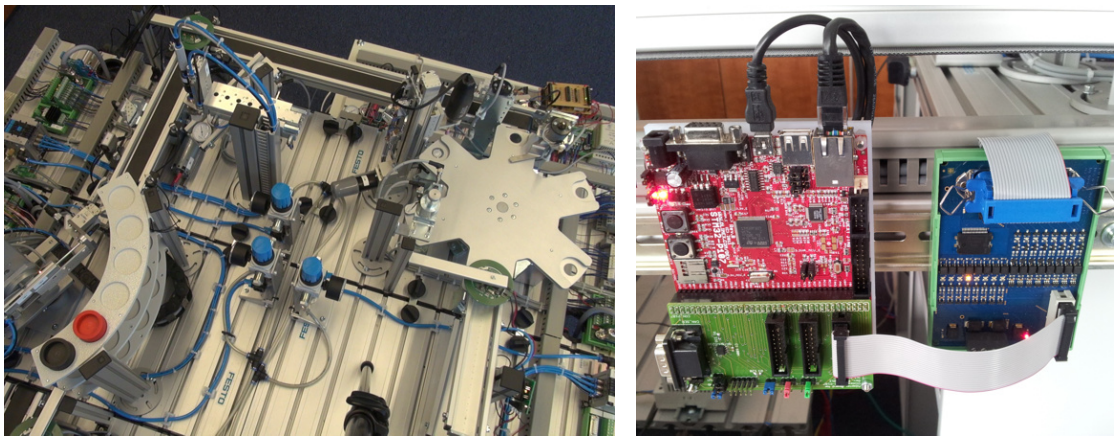


Figure 7.7.: Photos of real plant implementing bidirectional material flow: on the left, storage with black and red work piece, Pick & Place unit, conveyor belts, rotary plate with height sensor, drill and levers (in clockwise direction). On the right, microcontroller board (red) with custom adapter board (green, supports up to 48 digital I/Os, 7 analog inputs and 6 PWM generators) and 3.3 V/24 V interface board (blue, 16 digital I/Os per board).

- 9 hardware modules: conveyor belt, conveyor belt with barrier, 2 levers, rotary plate, height sensor, drill, pick & place module, robot arm storage.
- 3 types of sensor inputs: work piece height and color, storage rack occupancy.
- 4 ECUs of type Siemens S7-313C, where each controls a dedicated part of the plant (compare icons in Figure 7.6).

In order to represent all hardware modules, the following new model elements are introduced in addition to the ones introduced in Sections 4.4.1 and 7.5. For brevity, the formal modeling of those elements is omitted. Please refer to the accompanying tool MGSyn [for13] for details, which ships with the respective models.

- Behavioral interfaces:
  - robot-input, robot-eject: input a work piece from H to G and eject a work piece from G to H. The robot arm ejects the work piece by pushing it when the gripper is closed.
  - belt-move, belt-move-reverse: move a conveyor belt in (reverse) direction. Operation position J can be also reached using these actions.
  - mount-cap: mount a cap on a work piece located at operating position J.
- Predicates:
  - robot-input-located, robot-eject-located, belt-located belt-reverse-located: predicates that denote that two operating positions are connected via robot-input, robot-eject or a conveyor belt in forward or reverse direction (all read only).
  - pick-place-located: predicate that denotes where the pick & place unit is located (read only).
  - capped: predicate that denotes whether a cap is mounted on a work piece.

Task 4: Drill and Mount Cap on Small Red Work Pieces

Informal task description:

Drill and mount a cap on a work piece x_{wp1} initially located at operating position A if it is small and has red color and put it back to operating position A in any case. The robot arm is located at operating position G at beginning and end of the task[2].

[2] See the following video for the execution of a similar task on the real plant (with unconditional drilling): http://youtu.be/7p5EK52TgBs.

Formal task description:

$$\mathfrak{T}_4 = (\mathfrak{P}_0, \mathcal{C}_{\mathcal{I}4}, \sigma_4, g_4, \odot_4, \otimes_4, \eta_{\max 4}, d_4) \tag{7.4}$$

$$\mathcal{C}_{\mathcal{I}4} := \{\mathsf{at}(x_{\mathsf{wp1}}, \pi_{\mathsf{A}}), \mathsf{rack\text{-}occupied}(\pi_{\mathsf{A}}, x_{\mathsf{yes}}), \mathsf{in\text{-}robot}(\pi_{\mathsf{G}})\} \tag{7.5}$$

$$\begin{aligned} g_4 := {} & (\mathsf{color}(x_{\mathsf{wp1}}, x_{\mathsf{red}}) \vee \mathsf{color}(x_{\mathsf{wp1}}, x_{\mathsf{black}}) \vee \mathsf{color}(x_{\mathsf{wp1}}, x_{\mathsf{none}})) \\ & \wedge (\mathsf{color}(x_{\mathsf{wp2}}, x_{\mathsf{red}}) \Rightarrow ((\mathsf{height}(x_{\mathsf{wp1}}, x_{\mathsf{small}}) \vee \mathsf{height}(x_{\mathsf{wp1}}, x_{\mathsf{large}})) \\ & \quad \wedge (\mathsf{height}(x_{\mathsf{wp1}}, x_{\mathsf{small}}) \Leftrightarrow \mathsf{drilled}(x_{\mathsf{wp1}})) \\ & \quad \wedge (\mathsf{height}(x_{\mathsf{wp1}}, x_{\mathsf{small}}) \Leftrightarrow \mathsf{capped}(x_{\mathsf{wp1}})))) \\ & \wedge (\neg\mathsf{color}(x_{\mathsf{wp1}}, x_{\mathsf{red}}) \Rightarrow (\neg\mathsf{drilled}(x_{\mathsf{wp1}}) \wedge \neg\mathsf{capped}(x_{\mathsf{wp1}}))) \\ & \wedge \mathsf{at}(x_{\mathsf{wp1}}, \pi_{\mathsf{A}}) \wedge \mathsf{in\text{-}robot}(\pi_{\mathsf{G}}) \end{aligned} \tag{7.6}$$

Scenarios:

4a) **Centralized execution with 1 work piece:** Synthesize a single control program for the specification that remotely controls the individual ECUs.

4b) **Centralized execution with 2 work pieces:** Scenario 4a) with 2 work pieces.

4c) **Decentralized execution with 1 work piece:** Synthesize a dedicated control program for each ECU in the plant model.

4d) **Decentralized execution with 2 work pieces:** Scenario 4c) with 2 work pieces.

Evaluation results:

Table 7.3 summarizes the results for task $\mathfrak{T}_4$. Some scenarios require a lot of memory and hence the Java Virtual Machine is started with parameter *-Xmx3584m* (maximum 3.5 GB Java heap size) and the option *Allocate more memory for BDD (64-bit only)* is used in the MGSyn wizard (compare Figure 7.2). Some scenarios would also work with the default Java Heap Space of 3.0 GB, but suffer from bad performance due to extensive garbage collection and swapping.

In this plant, only a few control moves can be parallelized. The application of σ_{rp} to scenario 4b) yields a very good result of 36 % parallelized moves, but this is an exception. In this case, the solver generates a strategy that, when both work pieces are red, first moves them to the rotary plate, then puts one onto the rotary plate to probe its height and then puts the second one on one of the free operating positions on the rotary plate in order to avoid one work piece blocking the other. Then it continues with the second work piece.

This evaluation reveals a couple of issues with this approach:

- When a work piece is not red, the control program might trigger useless moves of the robot arm, especially when parallelization is enabled. This happens even if quantitative synthesis is used, because the specified cost bound applies to the worst case, which is the probing of a work piece's height, its drilling and mounting of a cap. However, if the work piece is not red, it "only" needs to be put back into the storage, leaving a lot of "cost" to waste. Since the only guarantee is

Table 7.3.: Synthesis results for task $\mathfrak{T}_4$ in the bidirectional example. Distributed synthesis uses token-based replication of the full state. Times are best ones out of two runs on a 3.2 GHz system with 8 GB of RAM.

Experiment	σ_4	d_4	$\eta_{\max 4}$	$\odot_4$	$\otimes_4$	**WC moves**	**Num. vars**	**Synthesis time (sec)**
4a) Centralized execution with 1 work piece	σ_{r}	1	∞	N/A	N/A	28	51	2.9
	σ_{rp}	2	∞	N/A	N/A	28[2]	51	16.2
	σ_{rc}	1	54	sum	N/A	inf.[1]	57	3.5
	σ_{rc}	1	55	sum	N/A	28	57	3.5
	σ_{rcp}	2	54	sum	sum	inf.[1]	57	27.8
	σ_{rcp}	2	55	sum	sum	28[3]	57	28.5
4b) Centralized execution with 2 work pieces	σ_{r}	1	∞	N/A	N/A	55	51	6.1
	σ_{rp}	2	∞	N/A	N/A	44[4]	51	21.3
	σ_{rc}	1	105	sum	N/A	inf.[1]	58	46.5
	σ_{rc}	1	106	sum	N/A	55	58	61.6
	σ_{rcp}	2	105	sum	sum	inf.[1]	58	309.6
	σ_{rcp}	2	106	sum	sum	55[3]	58	425.0
4c) Decentralized execution with 1 work piece	σ_{r}	1	∞	N/A	N/A	35	54	3.2
	σ_{rp}	2	∞	N/A	N/A	35[3]	54	14.5
	σ_{rc}	1	54	sum	N/A	inf.[1]	60	3.4
	σ_{rc}	1	55	sum	N/A	35	60	3.9
	σ_{rcp}	2	54	sum	sum	inf.[1]	60	13.5
	σ_{rcp}	2	55	sum	sum	35[3]	60	14.3
4d) Decentralized execution with 2 work pieces	σ_{r}	1	∞	N/A	N/A	68	54	6.9
	σ_{rp}	2	∞	N/A	N/A	68[3]	54	16.6
	σ_{rc}	1	105	sum	N/A	inf.[1]	61	66.9
	σ_{rc}	1	106	sum	N/A	68	61	85.5
	σ_{rcp}	2	105	sum	sum	inf.[1]	61	81.6
	σ_{rcp}	2	106	sum	sum	68[3]	61	106.7

[1] Infeasible (i.e., no solution) due to the cost bound being too restrictive.
[2] 7 of the 28 moves on the WC path are parallelized and cause useless side effects.
[3] None of moves on the WC path are parallelized.
[4] 16 of the 44 moves on the WC path are parallelized.

that the cost bound is not exceeded, this useless triggering of actuations is valid from the point of view of the model. The root cause is that quantitative synthesis only optimizes the worst-case path; shorter paths may contain superfluous or redundant executions of behavioral interfaces. Hence, increasing the cost for the respective action does not solve the problem. This behavior could be prevented by always forcing the solver to select the path with the least cost, which would most probably increase synthesis time significantly. Hence, this behavior is the price to pay for the trade-off between synthesis time and optimality of the solver in the presence of partial knowledge.

- Notice that decentralized quantitative synthesis with parallelization is clearly faster than centralized quantitative synthesis with parallelization. This result is unexpected, but can be explained by the fact that ordering of variables in the internal data structures of the game-based solver has a large effect on the synthesis time. The solver internally uses BDD as data structures. The efficiency of encoding Boolean terms in BDDs depends on the ordering of variables in the decision diagram, which in turn depends on the sorting of the elements in the PDDL specification. In the case at hand, the synthesis time differs roughly by the factor 4, which may be a hint that four times as many iterations are performed for the centralized control program, because variables are less efficiently encoded than in the decentralized case. Reordering the variables in order to optimize the encoding would require an adaptation in the game-based solver.

7.7. Summary

This chapter demonstrates how the workflow presented in this work is implemented in a model-driven development tool and provides evaluation results for small and medium scale applications. The evaluation scenarios represent typical automation tasks found in industry.

MGSyn is designed as a plug-in to the well-known Eclipse platform and inherits the concepts of operation from it. The hardware, plant and task model is specified in a tree-based editor. Extensive constraint checking ensures that meaningful warnings and error messages are produced in case the model is inconsistent. Synthesized control programs are either simulated on the development machine, remotely control the ECUs in the plant or run directly on the ECUs of the plant.

In conclusion, evaluation results show that the described approach leads to acceptable synthesis time on modern PC hardware for synthesizing centralized and decentralized control programs if the number of state space variables is reasonably small; a value of about 60 seems to be still acceptable. Larger numbers of variables may cause synthesis time to take several minutes or longer and pose more memory requirements.

The optimality of the synthesized control program is partially influenced by quantitative synthesis. However, this approach only allows optimization of plays with a worst-case number of moves for a given specification and is hence not a guarantee for an optimized or even optimal control program. In case the control program is not suitable, forcing the solver to produce a different result requires explicit elimination of

undesirable moves by adapting the model accordingly. Hence, the approach is recommended for scenarios where cost efficient (re-)synthesis of a control program outperforms the need to obtain an optimized or optimal control program. Examples are small or medium size enterprises that cannot afford experts in mechatronics who could manually build or adapt the control program. It is hence not a replacement for traditional workflows, but an alternative with certain restrictions.

Advantages of the approach include that it is correct by construction with respect to the input model, that the formal model is generic enough to serve for other automated tasks and that it facilitates the development of control software for decentralized systems.

CHAPTER 8

Conclusion

This thesis proposes a workflow for the synthesis of control software for specific types of industrial automation lines. First, a *formal model* of the plant, its hardware modules and capabilities is created in a *domain specific language*. The model also includes the behavior of the environment, which represents uncertainty in the production process, such as the detection of properties of a work piece only at runtime or the injection of faults. Second, a *task model* is specified that expresses the initial state and the goal state of the production process by describing the respective system state at both points in time. Subsequently, a *game-based solver* is invoked to *synthesize a centralized or multiple decentralized control programs* that achieve the functionality specified in the task model in the presence of the environment, i.e., the synthesized control program(s) reach the goal state from the initial state independent of the behavior of the environment. If no such strategy exists, the task is reported as infeasible. Finally, the synthesized control programs are *mapped* to the target platforms for simulation and execution.

The approach allows separating the traditional *bottom-up* development workflow into different stages, thus forming a *confluent* workflow:

- In the first stage, experts in mechatronics create a generic *hardware model* and implement primitive control functions that trigger certain actions in the plant. The hardware model is independent of a concrete automation task and serves as a kind of "driver library" for the real control programs. This is the "bottom-up" part of the workflow, because it forms the base for higher-level functionality.
- In the second stage, experts in industrial automation create a *plant model* that specifies the exact hardware modules and topology of a specific plant based on the previously defined hardware model.
- In the third stage, the task to perform is specified. This step can be performed with less or even no expert knowledge in mechatronics or control theory. The task model may contain a notion of cost that is used to select control programs

that offer certain performance guarantees. The last two steps are the "top-down" parts of the workflow, because they are formulated at an abstract level and lower-level details are automatically generated from those models.

The feasibility of the approach is demonstrated on both fictional as well as real automation systems. The real automation systems consist of parts that are also used in industrial production lines and hence model real production processes. The use cases that were analyzed in this work show that the time required for the synthesis of control programs is in the range of seconds and minutes and hence acceptable in the target domain.

The major contributions of this thesis are the definition of a formal domain specific modeling language to describe industrial automation lines and the tasks to perform on them, the ability to synthesize decentralized control programs for distributed execution on the *electronic control units (ECUs)* of the plant and a generic platform mapping approach to simulate and execute the synthesized control programs.

However, the following two limitations have been identified: first, specification of complex tasks requires more in-depth knowledge of the mathematical model behind this work and adding suitable constraints in the model might lead to unacceptably long synthesis times. Second, even when using quantitative synthesis, only the path with the worst-case cost in the play is optimized. Hence, unnecessary side effects may be present in plays that do not correspond to the worst-case cost.

Notice that the formal description of modular assembly lines is applicable to more than just control software synthesis. The presented model, or an extended version, could be used in automatic processes such as performance analyses, safety analyses, report generation or automatic documentation.

Future Work

The presented approach can be extended in various ways. The following list summarizes directions for future research.

- **Hierarchical synthesis:** Until now, synthesis is performed on a "flat" model, i.e., all information about the plant is considered for every run of the synthesis engine. This makes the approach less scalable, because the computational complexity of game solving is exponential in the number of state variables.

  In order to overcome this limitation, a hierarchical synthesis approach is meaningful: instead of solving the whole problem, (semi-)automatically split the task into smaller chunks and treat each chunk separately. Previously synthesized results from smaller chunks may be reused for later synthesis requests. This idea leads to a compositional synthesis approach.

  A similar approach can be used to automatically generate low-level primitive control functions: consider the actuation "transport of a work piece on a conveyor belt". By treating the transport problem as yet another synthesis problem that involves the actions of switching digital control signals on and off, a low-level control program implementing the primitive control function could be synthe-

sized. The triggering of this control program could then be used as a behavioral interface for a higher-level control program.

Likewise, the capabilities of a processing unit consisting of multiple modules could be determined once the model of that processing unit is complete and then suitable control actions synthesized for the unit, which are re-used in a higher-level synthesis problem as primitive control actions.

- **Non-cooperative environment:** Only sensor inputs have been considered as environment actions in the examples presented in this thesis. However, the environment could also inject faults into a system. Assuming a fault is temporary and we can recover from it, a suitable strategy can repeat the triggering of a part of the control program until it completes successfully. In order to embed this feature into the synthesized control programs, a "repeat until" kind of semantics is required. This can be realized with so-called *goal-or-loop* semantics: either the environment finally cooperates and hence the goal is reached successfully or the control program loops indefinitely. In case the fault is temporary, successful execution can hence be guaranteed. If using quantitative synthesis and the loop contains sensor triggerings, we need to ensure that their cost is zero; otherwise the environment immediately has a strategy to exceed the cost bound.

  A simple example is a conveyor belt: define an actuation that starts a conveyor belt in order to transport a work piece. Further define a sensor triggering that models the fact whether or not the work piece has arrived at the destination. By executing the sensor triggering in a loop, a wait condition for the arrival of the work piece at the destination can be achieved. Initial analyses of this approach have been performed in context of [CGB13].
- **Pipeline-based processing:** In production facilities, a pipeline-based approach is often used to process multiple work pieces at the same time in different areas of production. The current approach allows such processing, but requires the specification of the initial location, production goal and final location separately for every work piece. In order to better support pipeline-based processing, it should be possible to synthesize a control program that models the processing steps applied to a single work piece and then automatically parallelizing this execution pattern for use with multiple work pieces in different stages of production. The already implemented parallel task execution is a good base for implementing such an approach.
- **Generation of native *programmable logic controller (PLC)* control programs:** In order to natively execute synthesized control programs on PLCs, a suitable platform mapping layer is required. A promising approach seems to be the generation of control programs in *structured text (ST)*.
- **Generation of real-time visualization:** The information contained in the model can be used to generate a *graphical user interface (GUI)* that visualizes the current state of the system, similar to *supervisory control and data acquisition (SCADA)* systems. If the model is enhanced with topological information about the exact relative position of operating positions with respect to each other, a 2D or 3D visualization may be automatically generated.

APPENDIX A

Model Transformation: Token-based Ownership of Predicates

As indicated in Section 5.3.3, multiple strategies exist in order to replicate the system state between *electronic control units (ECUs)* during decentralized execution. At first sight, a reasonable granularity for replication seems to be to network each predicate individually and to generate according transfer actions during game-based solving. The purpose of this approach is to emulate the partial knowledge present in a distributed scenario in the problem that is analyzed by the game-based solver. Unfortunately, this approach does not produce feasible control programs. Nevertheless the approach is included here to show its advantages and disadvantages. The plant model is adapted as follows:

1. **Replication and ownership of predicates:** For every ECU, local copies of the predicates used on the respective ECU as derived in order to honor the fact that information about the state of the plant is not globally available in a distributed environment. This way, the solver will take transfer operations into account when generating the strategy. In addition, a predicate is added that indicates which ECU owns the most up-to-date value of the respective predicate is added for every original predicate that is read or written on at least two ECUs.

   For example, if the predicate height is identified as being read and/or written by two ECUs e1 and e2, then ECU-local variants named height-e1 and height-e2 are added and the original predicate is removed. In addition, the predicate height-owner $: \{\mathsf{e1}, \mathsf{e2}\} \rightarrow \mathbb{B}$ is added that is true for at most one ECUs[1]. The initial state is "all false", which means all ECUs know the correct value. Likewise, if predicate drilled is read and written only by e1, then it is replaced by an ECU-local variant drilled-e1. No drilled-owner predicate is needed in this case.
2. **Addition of ECU objects:** In order to represent which ECU currently "owns" a predicate, a corresponding object $x_{\hat{u}}$ is added for every ECU instance $\hat{u} \in \widehat{\mathcal{U}}_{\mathfrak{P}}$

[1]Notice that this predicate can profit from binary encoding as introduced later in Section 5.5.3.2.

in the same way than it is done in token-based replication of the full state from Section 5.3.3.1.

3. **Splitting of behavioral interfaces:** For every behavioral interface, determine whether it needs to be executed on different ECUs, and if yes, clone the behavioral interface in the same way than it is done in token-based replication of the full state from Section 5.3.3.1. Afterwards, the predicates read or written by the cloned behavioral interfaces are adapted to use the ECU-local copies available on the respective ECU.
4. **Addition of ownership and transfer operations:** In order to exchange state information between ECUs, introduce behavioral interfaces that transfer "ownership" of a predicate between ECUs. When ownership is transferred, the current predicate value is transferred as well. In general, the availability of such transfer actions between a pair of ECUs depends on the network topology available. In this work, a fully connected topology is assumed.

   First, a behavioral interface $\mathsf{b}_{\text{own-}v}$ for initially assigning the ownership to one of the ECUs is added for every ownership predicate v-owner (recall that the initial value of each ownership predicate is "all false", hence no specific ECU assignment):

$$\mathsf{b}_{\text{own-}v} := ((\mathsf{a}_{v_0}, \mathsf{a}_{\text{ecu}}), \mathcal{C}_{\mathcal{P}\mathsf{b}_{\text{own-}v}}, \mathcal{E}_{\mathsf{b}_{\text{own-}v}}, \emptyset, \emptyset, \bot, 0, 0) \tag{A.1}$$

$$\mathcal{C}_{\mathcal{P}\mathsf{b}_{\text{own-}v}} := \{\neg v\text{-owner } x_{\hat{u}} \mid \hat{u} \in \widehat{\mathcal{U}}_{\mathfrak{P}}\} \tag{A.2}$$

$$\mathcal{E}_{\mathsf{b}_{\text{own-}v}} := \{v\text{-owner } \mathsf{a}_{\text{ecu}}\} \tag{A.3}$$

$$\mathcal{B}_{\mathfrak{H}} := \mathcal{B}_{\mathfrak{H}} \cup \{\mathsf{b}_{\text{own-}v}\} \tag{A.4}$$

   If v has at most one parameter, the signature of $\mathsf{b}_{\text{own-}v}$ is $\mathcal{X}_{\text{ecu}} \to \mathbb{B}$ (i.e., a_{v_0} is omitted, compare equation (A.1)), otherwise an additional parameter whose name and type corresponds to the first parameter of v is present. In the latter case, predicate values are owned depending on the value of the first parameter, further modularizing variable replication. Consider for example the "at" predicate: if the location of x_{wp1} is only relevant on ECU e1 and the location is of x_{wp2} is only relevant on ECU e2, using an ownership predicate with signature at-owner: $\mathcal{W} \times \mathcal{X}_{\text{ecu}} \to \mathbb{B}$ avoids the need to transfer the locations, while using an ownership predicate of at-owner: $\mathcal{X}_{\text{ecu}} \to \mathbb{B}$ requires the transfer. The preconditions in equation (A.2) specify that v-owner is "all false" and the effect in equation (A.3) designates the ECU specified in a_{ecu} as first owner.

   Second, a behavioral interface $\mathsf{b}_{\text{transfer-}v}$ for transferring the ownership from one ECU to another is added for every ownership predicate v-owner:

$$\mathsf{b}_{\text{transfer-}v} := ((\mathsf{a}_{v_0}, \mathsf{a}_{\text{ecu1}}, \mathsf{a}_{\text{ecu2}}), \mathcal{C}_{\mathcal{P}\mathsf{b}_{\text{transfer-}v}}, \mathcal{E}_{\mathsf{b}_{\text{transfer-}v}}, \emptyset, \emptyset, \hat{p}_{\text{transfer-}v}, 0, 0) \tag{A.5}$$

$$\mathcal{C}_{\mathcal{P}\mathsf{b}_{\text{transfer-}v}} := \{v\text{-owner } \mathsf{a}_{\text{ecu1}}\} \tag{A.6}$$

$$\mathcal{E}_{\mathsf{b}_{\text{own-}v}} := \{\neg v\text{-owner } \mathsf{a}_{\text{ecu1}}, v\text{-owner } \mathsf{a}_{\text{ecu2}}\} \tag{A.7}$$

$$\mathcal{B}_{\mathfrak{H}} := \mathcal{B}_{\mathfrak{H}} \cup \{\mathsf{b}_{\text{transfer-}v}\} \tag{A.8}$$

   If v has at most one parameter, the signature of $\mathsf{b}_{\text{transfer-}v}$ is $\mathcal{X}_{\text{ecu}} \times \mathcal{X}_{\text{ecu}} \to \mathbb{B}$ (i.e., a_{v_0} is omitted, compare equation (A.5)), otherwise an additional parameter

whose name and type corresponds to the first parameter of v is present. The *primitive control function (PCF)* $\hat{p}_{\mathsf{transfer}\text{-}v}$ implements the actual send and receive operations between the two ECUs, where the ECU specified by $\mathsf{a}_{\mathsf{ecu1}}$ is the source and $\mathsf{a}_{\mathsf{ecu2}}$ is the destination. Suitable functions are automatically provided via code generation approach as described in Chapter 6. Preconditions and effects are self-explanatory.

Notice that the cost for transfer operations is set to zero here in order not to affect quantitative synthesis. In case transfer operations are expensive, the cost may be set to a value larger than zero.

5. **Adaptation of initial conditions:** The initial conditions need to be adapted such that the initial state formulated over the global predicates is propagated to all replicated predicates. Hence, the initial state is assumed to be globally known.

   For every initial condition of a module $\mathcal{C}_{\mathcal{I}_\varphi}, \varphi \in \Phi_{\mathfrak{H}}$ and every initial condition of the task to execute $\mathcal{C}_{\mathcal{I}_\mathfrak{T}}$, clone all conditions that reference *replicated predicates* (i.e., predicates that are read or written by more than one ECU) $\mathcal{V}' \subseteq \mathcal{V}_{\mathfrak{H}}$ for every ECU that reads or writes them $\mathrm{RW}(\mathcal{V}')$ and adapt the individual conditions such that they use the locally available predicates on the respective ECU. Then remove the respective original initial conditions from the model, because they refer to predicates that do not exist anymore.

   Consider for example the initial task condition $\mathsf{height}(x_{\mathsf{wp1}}, x_{\mathsf{small}})$, then if the hardware model contains two ECUs $\mathsf{e1}$ and $\mathsf{e2}$, the initial task condition is replaced by a set of two initial task conditions: $\mathsf{height\text{-}e1}(x_{\mathsf{wp1}}, x_{\mathsf{small}})$ and $\mathsf{height\text{-}e2}(x_{\mathsf{wp1}}, x_{\mathsf{small}})$.

6. **Adaptation of goal specification:** Finally, the goal conditions of the goal specification $g_\mathfrak{T}$ need to be adapted such that the fact that the goal state is reached is correctly decided on all ECUs. The desired behavior is that the predicate values that are relevant for deciding whether the goal has been reached are replicated to all ECUs. This is achieved by demanding in the "global" goal condition that each replicated instance of a predicate that is part of the goal condition needs to have the same value.

   Assume the goal condition is represented in conjunctive normal form. For every clause in the goal condition, analyze if it contains replicated predicates $\mathcal{V}' \subseteq \mathcal{V}_{\mathfrak{H}}$. If this is the case, clone the clause for every ECU $\mathrm{RW}(\mathcal{V}')$ that reads or writes any of the replicated predicates in the clause and adapt the individual clauses such that they use the locally available predicates on the respective ECU. Then remove the original clause from the goal condition. For example, if the goal condition contains the clause

$$\mathsf{height}(x_{\mathsf{wp1}}, x_{\mathsf{small}}) \vee \mathsf{height}(x_{\mathsf{wp1}}, x_{\mathsf{large}}) \tag{A.9}$$

   and $\mathrm{RW}(\mathsf{height}) = \{\mathsf{e1}, \mathsf{e2}\}$ (i.e., the height of work pieces is read or written on ECUs $\mathsf{e1}$ and $\mathsf{e2}$), then the clause is replaced by two clauses of the following form:

$$\begin{aligned}&\big(\mathsf{height\text{-}e1}(x_{\mathsf{wp1}}, x_{\mathsf{small}}) \vee \mathsf{height\text{-}e1}(x_{\mathsf{wp1}}, x_{\mathsf{large}})\big) \wedge \\ &\big(\mathsf{height\text{-}e2}(x_{\mathsf{wp1}}, x_{\mathsf{small}}) \vee \mathsf{height\text{-}e2}(x_{\mathsf{wp1}}, x_{\mathsf{large}})\big)\end{aligned} \tag{A.10}$$

This approach "forces" the synthesis engine to add respective transfer operations.

This approach is scalable with respect to network bandwidth, but has two significant drawbacks: first, it leads to a high synthesis time, because the number of variables and hence the state space becomes very large even for a small number of ECUs. Second, this approach only networks the predicate values that are either part of the preconditions or (conditional) effects of the actions being executed. However, the guards for action execution most probably contain other predicates as well, whose values are not networked. Hence, this approach is unfortunately not feasible without additional post-processing.

Index

ECU instance, 52
ECU instance parameter assignment function, 52
GAVS+, 88
GAVS, 88
0-vertices, 26
1-vertices, 26

Action, 46
Actuation, 51
Arena, 26
Argument domain, 49
Assembly line, 2
Attractor computation, 28
Automatic, 2
Automatic programming, 24
Automation pyramid, 13
Automation system, 2

Behavioral interface, 49
Behavioral primitive, 34
Binary encoding, 95

Centralized control strategy, 74
Changeover time, 4
Check constraint, 23
Check rule, 77
Conditional effect, 50
Configurable module type parameter, 48
Configurable parameter, 51
Configurable parameter assignment function, 53
Confluent, 36
Consistency, 79
Control, 13
Control program, 4
Control system, 3
Control unit, 3
Controller, 27
Correct by construction, 3
Cost, 50
Cost bound, 55

Data acquisition, 13
Dead end, 26
Decentralized control program, 29
Decentralized control strategy, 74
Degree of automation, 2
Degree of parallelization, 55, 91
Discretization, 37
Distributed synthesis, 29
Domain knowledge, 32
Domain specific language, 21
Domain specification, 86

Edge relation, 26
Electronic control unit, 3
Environment, 27
Execution mode, 118
Expert, 32

Fieldbus, 3
Fluent, 90

Game, 26
Game theory, 25
Goal state specification, 54

Hardware module, 44
Hardware parameter, 110

Industrial automation, 2
Initial module condition, 52
Initial task condition, 54
Input channel, 51
Input signal, 48
Input signal assignment function, 52

Metamodel, 22
Model-based, 20
Model-driven, 20
Model-to-model transformation, 23
Model-to-text transformation, 24
Modular assembly line, 37
Module, 44
Module instance, 52
Module type, 48
Move set, 26

Object, 47
Object type, 47
Operation position, 48
Operation position instance, 52
Operation position mapping function, 52
Optimization, 13
Output channel, 51
Output signal, 48
Output signal assignment function, 53
Overlapping operating position, 53

Parallel composition operator, 55
Parallel execution, 54
Planning, 13
Planning domain definition language, 86
Play, 26
Player, 26
Plug and produce, 5
Precondition, 49
Predicate, 47
Primitive control function, 34
Problem specification, 86
Process model, 56
Program synthesis, 24

Replicated predicate, 141

Sensor response, 79
Sensor result condition, 50
Sensor triggering, 51
Sequential composition operator, 55
Sequential execution, 54
Simulation mode, 118
Smart factory, 16
Smart sensor, 16
Specification, 24
Spontaneous state change, 68
State, 45
State explosion problem, 94
Strategy, 25
Synthesis, 3, 27

Template, 24
Token, 26
Turn-based game, 25
Two-player game, 26

Unconditional effect, 49

Validation, 23

Winner, 26
Winning set, 26
Work piece, 8

Bibliography

[3S-14a] 3S-Smart Software Solutions GmbH. CODESYS – industrial IEC 61131-3 PLC programming, 2014. http://www.codesys.com/ [Online; accessed 5-July-2014].

[3S-14b] 3S-Smart Software Solutions GmbH. CODESYS Application Composer, 2014. http://www.codesys.com/products/codesys-engineering/application-composer.html [Online; accessed 5-July-2014].

[Abb08] Doug Abbott. *Embedded Linux Development Using Eclipse*. Elsevier Science, 2008.

[aca11] acatech National Academy of Science and Engineering. Cyber-physical systems: Driving force for innovation in mobility, health, energy and production. *acatech POSITION*, December 2011. http://www.acatech.de/fileadmin/user_upload/Baumstruktur_nach_Website/Acatech/root/de/Publikationen/Stellungnahmen/acatech_POSITION_CPS_Englisch_WEB.pdf [Online; accessed 9-October-2012].

[act12] actifsource GmbH. actifsource – model driven software development that simply works, 2012. http://www.actifsource.com/ [Online; accessed 22-February-2013].

[AGH06] Ken Arnold, James Gosling, and David Holmes. *The Java^{TM}Programming Language*. Addison-Wesley, 4th edition, 2006.

[AMPS98] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. In *Proc. System Structure and Control*. Elsevier, 1998.

[Bak04] Peter Baker. The adoption of innovative warehouse equipment. In *Logistics Research Network 2004 Conference Proceedings*, pages 25–35, 2004.

[Bal85] Robert Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, November 1985.

[Bar81] David William Barron. *Pascal: The Language and its Implementation*. Wiley, 1981.

[BBF+12] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Acacia+, a tool for LTL synthesis. In P. Madhusudan and Sanjit A. Seshia, editors, *Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012)*, volume 7358 of *Lecture Notes in Computer Science*, pages 652–657. Springer Berlin Heidelberg, Berkeley, CA, USA, July 2012.

[BCDG08] Francesco Basile, Pasquale Chiacchio, and Domenico Del Grosso. Modelling automation systems by UML and Petri nets. In *9th International Workshop on Discrete Event Systems (WODES 2008)*, pages 308–313, May 2008.

[BCG+97] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-Software Co-design of Embedded Systems. The POLIS Approach*. Kluwer Academic Publishers, 1997.

[BDF+04] David Basin, Yves Deville, Pierre Flener, Andreas Hamfelt, and Jørgen Fischer Nilsson. Synthesis of programs in computational logic. In *Program Development in Computational Logic*, pages 30–65. Springer, 2004.

[BGBK08] Simon Barner, Michael Geisinger, Christian Buckl, and Alois Knoll. EasyLab: Model-based development of software for mechatronic systems. In *Proceedings of the 4th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA 2008)*, pages 540–545, Beijing, China, October 2008.

[BGH+10] Simon Barner, Michael Geisinger, Jia Huang, Alois Knoll, Holger Bönicke, Christoph Ament, Jochen Mades, Reinhard Pittschellis, and Gerd Bauer. EasyKit - Eine allgemeine Methodik für die Entwicklung von Steuerungskomponenten. In Jürgen Gausemeier, Franz Ramming, Wilhelm Schäfer, and Ansgar Trächtler, editors, *Entwurf mechatronischer Systeme*, volume 272 of *HNI-Verlagsschriftenreihe*, pages 23–36, Paderborn, Germany, March 2010.

[BH07] Peter Baker and Zaheed Halim. An exploration of warehouse automation implementations: cost, service and flexibility issues. *Supply Chain Management: An International Journal*, 12(2):129–138, 2007.

[Böh12a] Tino M. Böhler. Industrie 4.0: Automatisierer halten Zukunft in ihren Händen. *Produktion – Technik und Wirtschaft für die deutsche Industrie*, June 2012. `http://www.produktion.de/automatisierung/industrie-4-0-automatisierer-halten-zukunft/` [Online; accessed 10-October-2012].

[Böh12b] Tino M. Böhler. Industrie 4.0 – Smarte Produkte und Fabriken revolutionieren die Industrie. *Produktion – Technik und Wirtschaft für die deutsche Industrie*, May 2012. `http://www.produktion.de/automatisierung/industrie-4-0-smarte-produkte-und-fabriken/` [Online; ac-

cessed 10-October-2012].

[BL67] J. Richard Büchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. Technical Report 14, Purdue University, Lafayette, Indiana, USA, September 1967.

[BL10] Christopher Brooks and Edward A. Lee. Ptolemy II – heterogeneous concurrent modeling and design in Java. Poster presented at the 2010 Berkeley EECS Annual Research Symposium (BEARS), February 2010. `http://chess.eecs.berkeley.edu/pubs/655.html` [Online; accessed 11-February-2013].

[BMW12] BMWi (Bundesministerium für Wirtschaft und Technologie / German Federal Ministry of Economics and Technology). AUTONOMIK für Industrie 4.0: Produktion, Produkte, Dienste im multidimensionalen Internet der Zukunft, October 2012. `http://www.autonomik40.de/_media/BMWi_Broschuere_Autonomik_WEB.pdf` [Online; accessed 13-November-2013].

[BR13] Ezio Bartocci and C. R. Ramakrishnan, editors. *Proceedings of the 20th International Symposium on Model Checking Software (SPIN 2013)*, volume 7976 of *Lecture Notes in Computer Science*, Stony Brook, NY, USA, July 2013. Springer.

[BSM+04] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework: a developer's guide*. Addison-Wesley, 2004.

[Buc08] Christian Buckl. *Model-Based Development of Fault-Tolerant Real-Time Systems*. Dissertation, Technische Universität München, München, Germany, 2008.

[Bur87] Steve Burbeck. Applications programming in Smalltalk-80: How to use model-view-controller (MVC), 1987. `http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html` [Online; accessed 22-February-2013].

[CBLK10] Chih-Hong Cheng, Christian Buckl, Michael Luttenberger, and Alois Knoll. GAVS: Game arena visualization and synthesis. *Automated Technology for Verification and Analysis*, pages 347–352, 2010.

[CGB13] Chih-Hong Cheng, Michael Geisinger, and Christian Buckl. Synthesizing controllers for automation tasks with performance guarantees. In *Proceedings of the 20th International SPIN Symposium on Model Checking of Software (SPIN 2013)*, volume 7976 of *Lecture Notes in Computer Science*, pages 154–159, Stony Brook, NY, USA, July 2013. Springer.

[CGR+12a] Chih-Hong Cheng, Michael Geisinger, Harald Ruess, Christian Buckl, and Alois Knoll. Game solving for industrial automation and control. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2012)*, May 2012.

[CGR+12b] Chih-Hong Cheng, Michael Geisinger, Harald Ruess, Christian Buckl, and Alois Knoll. MGSyn: Automatic synthesis for industrial automation. In

Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012), volume 7358 of *Lecture Notes in Computer Science*, pages 658–664, Berkeley, CA, USA, July 2012. Springer.

[Che12] Chih-Hong Cheng. *An Implementation for Algorithmic Game Solving and its Applications in System Synthesis*. Dissertation, Technische Universität München, München, Germany, 2012.

[CJG+11] Chih-Hong Cheng, Barbara Jobstmann, Michael Geisinger, Sarah Diot-Girard, Christian Buckl, Alois Knoll, and Harald Ruess. Optimizations for game-based software synthesis. Technical Report TR-2011-12, Verimag Research Report, August 2011. `http://www-verimag.imag.fr/TR/TR-2011-12.pdf` [Online; accessed 14-March-2013].

[CKLB11] Chih-Hong Cheng, Alois Knoll, Michael Luttenberger, and Christian Buckl. GAVS+: an open platform for the research of algorithmic game solving. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*, Lecture Notes in Computer Science. Springer, April 2011.

[CSV96] Alberto Cavallo, Roberto Setola, and Francesco Vasca. *Using MATLAB, SIMULINK, and Control System Toolbox: a practical approach*. The MATLAB curriculum series. Prentice Hall, 1996.

[CT02] Martin Christopher and Dennis R. Towill. Developing market specific supply chain strategies. *International Journal of Logistics Management*, 13(1):1–14, 2002.

[DF03] Jolyon Drury and Peter Falconer. *Building and Planning for Industrial Storage and Distribution*. Architectural Press, Oxford, 2nd edition, 2003.

[DJ91] Kofi Q. Dadzie and Wesley J. Johnston. Innovative automation technology in corporate warehousing logistics. *Journal of Business Logistics*, 12(1):63–82, 1991.

[Ecl13] Eclipse Foundation. Eclipse Modeling – Model Development Tools (MDT), 2013. `http://www.eclipse.org/uml2/` [Online; accessed 22-February-2013].

[Ecl14] Eclipse Foundation, Inc. Xtend – Modernized Java, section "Template Expressions", May 2014. `http://eclipse.org/xtend/documentation.html#templates` [Online; accessed 28-June-2014].

[EHN94] Kutluhan Erol, James Hendler, and Dana S Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Proceedings of the second International Conference on AI Planning Systems*, pages 249–254, 1994.

[EJL+03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2):127–144, January 2003.

[Ele14] Electroquip. FESTO FEC-FC640-FST 191450 Festo Controller, 2014. `http://www.electroquip.co.uk/festo-controller/`

festo-fec-fc640-fst-191450 [Online; accessed 5-July-2014].

[Fes12] Festo Didactic. MPS® the modular production system - learning systems, 2012. http://www.festo-didactic.com/int-en/learning-systems/mps-the-modular-production-system/ [Online; accessed 10-October-2012].

[FFMV08] Luca Ferrarini, Giuseppe Fogliazza, Giulia Mirandola, and Carlo Veber. Metamodeling techniques applied to the design of reconfigurable control applications. *EURASIP Journal on Embedded Systems*, 2008, 2008.

[Fle12] Marcel Flesch. Implementation of a system for sensor data fusion in thermal processes using a data centric middleware. Master's thesis, Technische Universität München, München, Germany, August 2012.

[FN72] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208, 1972.

[for13] fortiss GmbH. MGSyn – automatic synthesis for industrial automation, 2013. http://mgsyn.fortiss.org/ [Online; accessed 19-November-2013].

[FP04] Peter Fischer and Peter Palensky. "The importance of being certified": The role of conformance testing and certification of communication systems in building automation and control devices. In *Proceedings of the 7th IEEE AFRICON Conference in Africa*, volume 2, pages 1223–1227, September 2004.

[Fri11] Ronny Fritsche. Reducing set-up times for improved flexibility in high-mix low-volume electric drives production. In *1st International Electric Drives Production Conference (EDPC 2011)*, pages 74–77, September 2011.

[GBWK09] Michael Geisinger, Simon Barner, Martin Wojtczyk, and Alois Knoll. A software architecture for model-based programming of robot systems. In Torsten Kröger and Friedrich M. Wahl, editors, *Advances in Robotics Research – Theory, Implementation, Application*, pages 135–146, Braunschweig, Germany, 2009. Springer-Verlag Berlin Heidelberg.

[GC13] Michael Geisinger and Chih-Hong Cheng. Programm auf Knopfdruck? *Computer & Automation*, pages 32–35, August 2013. http://www.computer-automation.de/steuerungsebene/steuern-regeln/artikel/100262/ [Online; accessed 16-January-2014].

[Ger04] Jane Gerold. The factory of the future. *Automation World*, July 2004. http://www.automationworld.com/operations/factory-future [Online; accessed 21-September-2012].

[GHK+98] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. *PDDL – The Planning Domain Definition Language, Version 1.2*, 1998. Available at http://www.cs.yale.edu/homes/dvm/software/pddl.tar.gz [Online; accessed 13-March-2013].

[GL05] Alfonso Gerevini and Derek Long. Plan constraints and preferences in

PDDL3. Technical report, Department of Electronics for Automation, University of Brescia, Italy, 2005.

[GL13] Alfonso Gerevini and Derek Long. BNF description of PDDL3.0. Unpublished manuscript from the IPC-5 website, 2005. Available at `http://cs-www.cs.yale.edu/homes/dvm/papers/pddl-bnf.pdf` [Online; accessed 13-March-2013].

[Gro09] Richard C. Gronback. *Eclipse Modelig Project: a domain-specific language (DSL) toolkit*. Addison-Wesley, 2009.

[GS04] Brian J. Gough and Richard M. Stallman. *An Introduction to GCC: For the GNU Compilers gcc and g++*. Network theory manual. Network Theory, 2004.

[GTW03] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, logics, and infinite games: a guide to current research*, volume 2500. Springer, 2003.

[Har12] Matthias Harzheim. Firmware development for a redundant sensor system for thermal processes. Semesterarbeit, Technische Universität München, München, Germany, November 2012.

[Her13] Thomas Hergenhahn. LIBNODAVE, a free communication library for Simatic S7 PLCs, 2013. `http://sourceforge.net/projects/libnodave/` [Online; accessed 12-November-2013].

[HET+06] Jörg Hoffmann, Stefan Edelkamp, Sylvie Thiébaux, Roman Englert, Frederico Liporace, and Sebastian Trüg. Engineering benchmarks for planning: the domains used in the deterministic part of IPC-4. *Journal of Artificial Intelligence Research*, 26(1):453–541, 2006.

[HH12] Jana Hlubeňová and Daniel Hlubeň. Algorithm for selection of simulation software. *Advanced Materials Research*, 463–464:1077–1080, February 2012.

[HHK01] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the First International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science*, 2211:166–184, 2001. `http://embedded.eecs.berkeley.edu/giotto/` [Online; accessed 11-February-2013].

[IBM12] IBM Corporation. The Rational Rhapsody family from IBM – collaborative systems engineering and embedded software development, 2012. `http://public.dhe.ibm.com/common/ssi/ecm/en/rab14010usen/RAB14010USEN.PDF` [Online; accessed 22-February-2013].

[IEC03a] International Standard IEC 61131: Programmable Controllers, Part 3: Programming Languages, January 2003.

[IEC03b] International Standard IEC 61804-1: Function Blocks (FB) for Process Control, Part 1: Overview of System Aspects, October 2003.

[IEC03c] International Standard IEC 62264: Enterprise-Control System Integration, Part 1: Models and Terminology, March 2003.

[IEC05a] International Standard IEC 19502: Information Technology – Meta Object Facility (MOF), November 2005.

[IEC05b] Publicly Available Specification IEC PAS 62407: Real-time Ethernet Control Automation Technology (EtherCAT^{TM}), June 2005. [Withdrawn].

[IEC06] International Standard IEC 60050: International Electrotechnical Vocabulary, Part 351: Control Technology, June 2006. `http://www.electropedia.org/` [Online; accessed 16-August-2012].

[IEC07] International Standard IEC 61158: Digital Data Communication for Measurement and Control – Fieldbus for use in Industrial Control Systems, December 2007.

[IEC10] International Standard IEC 61784: Industrial Communication Networks – Profiles, Part 1: Fieldbus Profiles, July 2010.

[IEC12] Final Draft International Standard IEC 61499-1: Function Blocks, Part 1: Architecture, August 2012.

[IET68] IETF RFC 20 (ANSI X 3.4-1968), October 1968. `http://tools.ietf.org/html/rfc20` [Online; accessed 28-February-2013].

[IL01] Frank Iwanitz and Jürgen Lange. *OLE for process control: fundamentals, implementation, and application*. Hüthig, 2001.

[IMB12] Klaus Irrgang, Uwe Meiselbach, and Gerd Bauer. Sensoren für die thermische Messfehlerkorrektur. *Mikroproduktion*, 2012(2):64–67, February 2012. `http://www.efm-systems.de/fileadmin/digiraster/dokumente/MIKROPRODUKTION_2_12_efm-systems_online.pdf` [Online; accessed 21-September-2012].

[IMS11] IMS. Ims research, 2011.

[ISI05] ISIS (Institute for Software Integrated Systems). MetaGME 2000: metamodeling environment, January 2005. `http://w3.isis.vanderbilt.edu/projects/gme/meta.html` [Online; accessed 28-June-2014].

[Jan07] David Janin. On the (high) undecidability of distributed synthesis problems. In Jan Leeuwen, Giuseppe F. Italiano, Wiebe Hoek, Christoph Meinel, Harald Sack, and František Plášil, editors, *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2007)*, volume 4362 of *Lecture Notes in Computer Science*, pages 320–329. Springer-Verlag, 2007.

[JCC97] Mu Der Jeng, Shih Wei Chou, and Chi Liang Chung. Performance evaluation of an IC fabrication system using Petri nets. In *IEEE International Conference on Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation*, volume 1, pages 269–274, October 1997.

[JGWB07] Barbara Jobstmann, Stefan Galler, Martin Weiglhofer, and Roderick Bloem. Anzu: a tool for property synthesis. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 258–262. Springer Berlin Heidelberg, Berlin, Germany, July

2007.

[Jon06] Clarence T. Jones. *STEP 7 in 7 Steps: A Practical Guide to Implementing S7-300/S7-400 Programmable Logic Controllers*. Brilliant-Training, 2006.

[KD97] Jongwook Kim and Alan A. Desrochers. Modeling and analysis of semiconductor manufacturing plants using time Petri net models: COT business case study. In *IEEE International Conference on Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation*, volume 4, pages 3227–3232, October 1997.

[KHJ+99] Yoram Koren, Uwe Heisel, Francesco Jovane, Toshimichi Moriwaki, G. Pritschow, G. Ulsoy, and H. Van Brussel. Reconfigurable manufacturing systems. *Annals of the CIRP - Manufacturing Technology*, 48(2):527–540, 1999.

[KKB13] Nadine Keddis, Gerd Kainz, and Christian Buckl. Towards adaptable manufacturing systems. In *Proceedings of the 2013 IEEE International Conference on Industrial Technology (ICIT 2013)*, February 2013.

[KKK84] Norihisa Komoda, Kazuo Kera, and Takeaki Kubo. An autonomous, decentralized control system for factory automation. *IEEE Computer*, 17(12):73–83, December 1984.

[KL92] Asawaree Kalavade and Edward A. Lee. Hardware / software co-design using Ptolemy – a case study. In *Proceedings of the First International Workshop on Hardware/Software Codesign*, Estes Park, Colorado, USA, September 1992.

[KLW11] Henning Kagermann, Wolf-Dieter Lukas, and Wolfgang Wahlster. Industrie 4.0: Mit dem Internet der Dinge auf dem Weg zur 4. industriellen Revolution. *VDI Nachrichten*, April 2011. `http://www.vdi-nachrichten.com/artikel/-/52570` [Online; accessed 9-October-2012].

[Kor10] Yoram Koren. *The Global Manufacturing Revolution – Product-Process-Business Integration and Reconfigurable Systems*. John Wiley & Sons, June 2010.

[Kov11] Daniel L. Kovacs. Complete BNF description of PDDL 3.1 (completely corrected). `http://www.plg.inf.uc3m.es/ipc2011-deterministic/attachments/Resources/kovacs-pddl-3.1-2011.pdf` [Online; accessed 30-November-2013], 2011.

[KR88] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, March 1988.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, March 2008.

[LLJ04] Chang-Pin Lin, Yi-Pin Lin, and Mu Der Jeng. Design of intelligent manufacturing systems by using UML and Petri net. In *2004 IEEE International Conference on Networking, Sensing and Control*, volume 1, pages 501–506, March 2004.

[MCF03] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Model-driven

development. *IEEE Software*, pages 14–18, September 2003.

[Met13] MetaCase. MetaEdit+ Domain-Specific Modeling (DSM) environment, 2013. http://www.metacase.com/products.html [Online; accessed 22-February-2013].

[MJNT00] R. Mason-Jones, B. Naylor, and D.R. Towill. Engineering the leagile supply chain. *International Journal of Agile Management*, 2(1):54–61, 2000.

[Mod06] Modbus-IDA. Modbus Application Protocol Specification V1.1b, December 2006. http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf [Online; accessed 4-September-2012].

[MS12] PParthasarathy Madhusudan and Sanjit A. Seshia, editors. *Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012)*, volume 7358 of *Lecture Notes in Computer Science*, Berkeley, CA, USA, July 2012. Springer.

[MUK00] Mostafa G. Mehrabi, A. Galip Ulsoy, and Yoram Koren. Reconfigurable manufacturing systems: key to future manufacturing. *Journal of Intelligent Manufacturing*, 11(4):403–419, 2000. http://deepblue.lib.umich.edu/bitstream/handle/2027.42/46513/10845_2004_Article_268791.pdf [Online; accessed 20-September-2014].

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[MW03] Swarup Mohalik and Igor Walukiewicz. Distributed games. In *Proceedings of the 23rd International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'03)*, volume 2914 of *Lecture Notes in Computer Science*, pages 338–351. Springer-Verlag, 2003.

[Nan98] Andreas Nann. *Prozeßdatenerfassung und -verarbeitung mit Siemens WinCC*. Dissertation, Fachhochschule für Technik und Wirtschaft, Offenburg, Germany, 1998.

[NB04] Stuart Naish and Peter Baker. Materials handling: fulfilling the promises. *Logistics and Transport Focus*, 6(1):18–26, 2004.

[NF70] Nils J. Nilsson and Richard E. Fikes. STRIPS: A new approach to the application of theorem proving to problem solving. Technical Report 43, Stanford Research Institute, California, USA, October 1970.

[OLK93] B. Orlik, J. Langfermann, and J. Kasting. Microcontroller-based invertors including a decentralized process control. In *Fifth European Conference on Power Electronics and Applications*, volume 6, pages 150–155, September 1993.

[OMG07] OMG (Object Management Group). OMG Data Distribution Service for Real-time Systems, version 1.2, January 2007. http://www.omg.org/spec/DDS/1.2/PDF [Online; accessed 5-August-2013].

[OMG11a] OMG (Object Management Group). OMG Meta Object Facility (MOF) Core Specification, version 2.4.1, August 2011. http://www.omg.org/spec/

MOF/2.4.1/PDF [Online; accessed 10-October-2012].

[OMG11b] OMG (Object Management Group). OMG Unified Modeling Language™ (OMG UML), Infrastructure, version 2.4.1, August 2011. http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF [Online; accessed 21-September-2012].

[OMG11c] OMG (Object Management Group). OMG Unified Modeling Language™ (OMG UML), Superstructure, version 2.4.1, August 2011. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF [Online; accessed 21-September-2012].

[OMG11d] OMG (Object Management Group). UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, version 1.1, June 2011. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF [Online; accessed 21-September-2012].

[OMG12a] OMG (Object Management Group). Information Technology - Object Management Group Object Constraint Language (OCL), version 2.3.1, April 2012. http://www.omg.org/spec/OCL/ISO/19507/PDF [Online; accessed 14-August-2013].

[OMG12b] OMG (Object Management Group). OMG Systems Modeling Language (OMG SysML™), version 1.3, June 2012. http://www.omg.org/spec/SysML/1.3/PDF [Online; accessed 10-October-2012].

[Ped89] Edwin P. D. Pednault. Adl: exploring the middle ground between strips and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 324–332, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of Reactive(1) designs. In E. Allen Emerson and Kedar S. Namjoshi, editors, *7th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer Berlin Heidelberg, 2006.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1989)*, pages 179–190, New York, NY, USA, 1989. ACM.

[PR90] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, volume 2, pages 746–757, October 1990.

[Pro13] Promotorengruppe Kommunikation der Forschungsunion Wirtschaft – Wissenschaft. Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0. Abschlussbericht, acatech – Deutsche Akademie der Technikwissenschaften e.V., April 2013. http://www.bmbf.de/pubRD/Umsetzungsempfehlungen_Industrie4_0.pdf [Online; accessed 4-November-2013].

[PSW00] Raja Parasuraman, Thomas B. Sheridan, and Christopher D. Wickens. A

model for types and levels of human interaction with automation. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 30(3):286–297, May 2000.

[Rab69] Michael O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, pages 1–35, 1969.

[RIF13] RIF e.V. CIROS-Engineering: CIROS Studio, 2013. `http://www.ciros-engineering.com/en/products/virtual_engineering/ciros_studio/` [Online; accessed 27-February-2013].

[Ros92] Roni Rosner. *Modular synthesis of reactive systems*. PhD thesis, The Weizmann Institute of Science, Rehovot, Israel, April 1992.

[Sch12] Jochen Schlick. Cyber-physical systems in factory automation - towards the 4th industrial revolution. In *9th IEEE International Workshop on Factory Communication Systems (WFCS 2012)*, May 2012.

[Sel11a] Bran Selić. The theory and practice of modeling language design, 2011. `http://ecs.victoria.ac.nz/foswiki/pub/Events/MODELS2011/Material/MODELS_2011_T3-Selic.ModelingLanguages.Tutorial.updated.v2.pdf` [Online; accessed 10-October-2012].

[Sel11b] Bran Selić. The theory and practice of modeling language design for model-based software engineering—a personal perspective. In João Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 290–321. Springer, 2011.

[SGB+13] Stephan Sommer, Michael Geisinger, Christian Buckl, Gerd Bauer, and Alois Knoll. Reconfigurable industrial process monitoring using the CHROMOSOME middleware. In *Fifth International Workshop on Adaptive and Reconfigurable Embedded Systems (APRES 2013)*. ACM, April 2013.

[SGT+12] Thierry Le Sergent, Alain Le Guennec, François Terrier, Yann Tanguy, and Sébastien Gérard. SCADE System, a comprehensive toolset for smooth transition from Model-Based System Engineering to certified embedded control and display software, May 2012. `http://www.esterel-technologies.com/technology/WhitePapers/` [Online; accessed 22-February-2013].

[Sie11] Siemens AG. *Standards Compliance according to IEC 61131-3:2003-12 (2nd Edition)*, 2011. `http://support.automation.siemens.com/AT/llisapi.dll/csfetch/50204938/IEC_61131_Compliance.pdf` [Online; accessed 6-September-2012].

[Sie13] Siemens AG. Product guide for totally integrated automation, 2013. `http://www.industry.siemens.com/topics/global/en/tia/Documents/tia-product-guide-en.pdf` [Online; accessed 24-June-2014].

[Sie14a] Siemens AG. Compact CPUs – PLCs, 2014. `http://w3.`

siemens.com/mcms/programmable-logic-controller/en/simatic-s7-controller/s7-300/cpu/compact-cpus/ [Online; accessed 5-July-2014].

[Sie14b] Siemens AG. LOGO! modular basic variants - PLCs, 2014. w3.siemens.com/mcms/programmable-logic-controller/en/logic-module-logo/modular-basic-variants/ [Online; accessed 5-July-2014].

[SK00] Douglas C. Schmidt and Fred Kuhns. An overview of the Real-Time CORBA specification. *IEEE Computer*, 33(6):56–63, June 2000.

[SM06] Thomas Schulz and Dimos Mitkoudis. Feinplanung von Fertigungsaufträgen: Optimierung als Kernfunktion von Manufacturing Execution Systems. *PPS Management*, 11:52–55, 2006.

[SME09] SMErobot™ consortium. SMErobot™, 2009. http://www.smerobot.org/ [Online; accessed 5-August-2013].

[SME12] SMErobotics consortium. SMErobotics, 2012. http://www.smerobotics.org/ [Online; accessed 5-August-2013].

[SP03] Apostolos Syropoulos and Basil K. Papadopoulos. A survey of computer operating systems for industrial applications. http://obelix.ee.duth.gr/~apostolo/Articles/rtos.pdf [Online; accessed 21-September-2012], April 2003.

[Spa13] Sparx Systems Pty Ltd. Enterprise Architect – UML Design Tools and UML CASE tools for software development, 2013. http://www.sparxsystems.com/products/ea/ [Online; accessed 22-February-2013].

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 4th edition, May 2013.

[SV06] Thomas Stahl and Markus Völter. *Model-driven software development*. John Wiley & Sons Chichester, 2006.

[SV13] Natasha Sharygina and Helmut Veith, editors. *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013)*, volume 8044 of *Lecture Notes in Computer Science*, Saint Petersburg, Russia, July 2013. Springer.

[The13] The MathWorks, Inc. Simulink – simulation and model-based design, 2013. http://www.mathworks.com/products/simulink/ [Online; accessed 22-February-2013].

[Tho84] Lyn C. Thomas. *Games, theory, and applications*. Ellis Horwood/John Wiley, Chichester, London, 1984.

[Tho05] Jean-Pierre Thomesse. Fieldbus technology in industrial automation. *Proceedings of the IEEE*, 93(6):1073–1101, June 2005.

[TK06] Jeffrey Travis and Jim Kring. *LabVIEW for Everyone: Graphical Programming Made Easy and Fun*. National Instruments Virtual Instrumentation Series. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition, 2006.

[Zho98] MengChu Zhou. Modeling, analysis, simulation, scheduling, and control of semiconductor manufacturing systems: A Petri net approach. *IEEE Transactions on Semiconductor Manufacturing*, 11(3):333–357, August 1998.