Universität Augsburg

ALSACE Memo

On the needs for specification and verification of collaborative and concurrent robots, agents and processes

Martin E. Müller

Report 2015-03 September 2015

Institut für Informatik
D-86135 Augsburg

ALSACE:
Algebraic and Logic Specification and Analysis of Concurrent Environments

Martin E. Müller
University of Augsburg, Dept. Computer Science
Technical Report

September 2015

Abstract

This report summarises and integrates two different tracks of research for the purpose of envisioning and preparing a joint research project proposal.
Soft- and hardware systems have become increasingly complex and act "concurrently", both with respect to memory access (i.e. information flow) and computational resources (i.e. "services"). The software development metaphor of cloud-storage, cloud-computing and service-oriented design has been anticipated by artificial intelligence (AI) research at least 30 years ago (parallel and distributed computation already dates back to the 1950's and 1970s). What is known as a "service" today is what in AI is known as the capability of an *agent*; and the problem of information flow and consistency has been a headstone of information processing ever since. Based on a real-world robotics application we demonstrate how an increasingly abstract description of collaborating or competing agents correspond to a set of concurrent processes.
In the second part we review several approaches to the theory of concurrent systems. Based on the different kinds of program semantics we present corresponding logical and algebraic means for the description of parallel processes and memory access. It turns out that Concurrent Kleene Algebra (CKA) and its related graphlet metaphor appears to deliver a one-to-one matching formal description of the module structures developed in the first part. The problem of snapshotting system states in order to receive (partial) traces of a running system seems to be well describable by a Temporal Logic of Actions (TLA). Finally, the different types of subsystems and their mutual requirements such as exclusiveness etc. seem to be best describable in a separation-logic like approach.
We conclude with a list of research questions detailing some of the many promising issues raised in the report.

Contents

Preliminary Remark

This is a *memorandum* — a collection of premature work.

It summarizes and refers to a large set of related research. This memo does not follow a strict notational convention which is due to the many different formalisms and their respective idioms.

This article includes a large number of simple but clear and vivid examples to generate an intuitive understanding of each topic's importance for our research proposal. Formal details of the theories that we refer to are intentionally omitted so as not to veil the "big picture" of the article.

This memo is not to state or even prove a set of well-defined theorems. The purpose is rather to identify relevant research directions and pave the way towards finding a suitable formalism and then formulate and examine the phenomena of specifications and verification of collaborative and concurrent robots, agents and processes.

Chapter 1

Robots, Programs, Processes

In this chapter, we give an informal introduction in the general setting of our research. We show that robots (or agents), programs and processes all suffer from similar problems concerning information flow—the only difference is what one considers as "information" or "token" that is passed from one processing "unit" to another.

1.1 Data, Programs, Computers

Concurrent and distributed data and information processing have become key technologies in computer science. The early steps of informatics were based on batch-organized computation on locally stored data. They were followed by mainframe architectures with one central computing unit and many time-sharing client terminals. With the advent of workstations, computational power became distributed in the sense that processor time was rather shared between processes than between users. Also, workstations provided local storage.

The specification and analysis of linear programs with exclusive memory access allowed verification of sequential processes, [27, 29]. After single processor, single process/thread, and local storage computers, first storage became distributed; i.e. a combination of file servers and local storage systems were available over a network for exclusive or concurrent access. With that many computational units connected to each other, where each of it was capable of processing multiple processes, the last step was to distribute the computational resources so that processes could be delegated to different manchines, should the local processor not provide enough computational power. The most recent fundamental change was to move both computational resources and data into the "cloud".

Of course, many different approaches have evolved in between these six "generations" of computing paradigms; some of them silently went extinct (massive parallel local computation, transputers, RISC), others were incorporated into new paradigms (preemptive/collaborative timesharing, ARM, FPGA) or were re-discovered in later generations

(thin clients, server farms providing thousands of processor nodes for massively parallel, distributed processing).

1.2 Products, Fabrication, Factories

Similar to the evolution of informatics, the methods of production (in an industrial sense) have changed. The individual artisan manufactured a product from raw material; one product after another. This method may have evolved into an "episodic" workflow where several products were produced "simultaneaously" by repeating an episode of the process for each item until moving to the next sequence of production steps. A natural optimization of this procedure is to run all the episodes in parallel by several workers, each one specialising in one production episode. Since every episode requires supply, in- and output of each episode needs to be passed sequentially, and full productivity can only be achieved by external (global) timing. The ultimate result of this process refinement led to assembly lines where the batch size of passed products is exactly one item and each episode is broken down into as few steps as possible so as to maximise the global timing frequency.
Other important inventions of industrial production are platform or module designs where components of complex products could be mutually exchanged: one component could be used in several models and one and one model could be built using different components. This method offered "production buffers" at the price of storage required for the modules in stock. Also, it became more likely for some workshops specialized to a certain episode within the production processes to idle because either their output storage was full or the required supply was exhausted.

The idea behind "industry 4.0" is that

> [...] businesses will establish global networks that incorporate their machinery, warehousing systems and production facilities [...]. [They] comprise smart machines, storage systems and production facilities capable of autonomously exchanging information, triggering actions and controlling each other independently. [...] Smart products are uniquely identifiable, may be located at all times and know their own history, current status and alternative routes to achieving their target state. [39]

One needs to understand that products knowing their own history are the key issue here: it is not the production plan that organizes the production processes but it is the product-to-be that triggers production processes to act upon them. Hence, the specific properties mentioned in the context of such production paradigms are interoperability, decentralization, real-time capability, service orientation, and modularity, [25]. In fact, it is a one-to-one correspondence to what we identified as "cloud computation" in the previous section.

1.3 Glitches, Faults and Failure

A *glitch* is some kind of local irregularity that does not (significantly) impair its surrounding processes and that can locally be repaired and does not require or cause a stir. A *faulty process* delivers products that do not conform to the specification and hence fail a quality test or cause a further fault or failure when ignored. Hence, faults can result in significant impairments of both the production process itself and the final product. They do cause a stir as they are recognised externally, but it usually only requires local repair to remove them. In order to avoid further faults or follow-up glitches (by trying to "adapt" to faulty supplies) the fault and its correction must be announced and documented. Ideally, glitches should be announced and documented as well but since their impact is closely confined and they are debugged before causing external stir, they usually remain undocumented.
Failure is some kind of a global fault. It occurs when the entire system and usually all of its components come to a halt because one or several parts produced or encountered a faulty result. Failure is easy to detect as it usually causes immediate breakdown. But it is not as easy to repair because the causing fault (or even glitch) needs to be identified. Undocumented glitches are nearly impossible to detect, while faults can be detected (and even avoided) by verifying the processes against their specification. The latter are hard to come by when not or only insufficiently documented, and, in the worst case, a "dead" system cannot be analysed at all for this would require the performance of the *running* system.
The problem of glitches and faults leading to failure is well known, as they may result in fatalities (Therac-25, Chinook engine control, Chinese Airbus A300 crash, Airbus 320 airshow crash, Iran Air Flight 655 shooting by USS Vincennes, London Ambulance Service Dispatch software, etc) critical situations (Intel bugs, Ariane 5 satellite launcher, H.M.S. Sheffield sinking after Exocet misclassification and Aegis problems, numerous manned and unmanned space missions, telecommunication breakdown, nuclear powerplant incidents, etc) or huge economic loss (Y2K, Airline reservation software and baggage routing, embedded automotive software, public transport breakdown, etc).

1.4 Programs and programming languages

Basically all mentionend generations of paradigms of computing correspond to programming paradigms, where it is not always obvious whether the former inspired the latter or vice versa (sometimes, they were developed in close cooperation but usually became extinct due to their proprietry character, e.g. LISP and the LISP machine or occam and Transputers).
The first and simplest programming languages are GOTO languages with their equivalent WHILE programs already gaining a benefit by allowing to easily specify iterative algorithms. Any such algorithms performs all its actions on a memory using variables that are globally accessible and, due to the purely sequential execution of the program, can be

safely read and changed at any discrete point in time of execution. But the bigger the problems, the more complicated the algorithm and the resulting program codse that grew into large, monolithic pieces of specialised software with many redundancies in software development.
Replacing iterative concepts by recursive ones gave rise to the question of how to manage different variable instantiations during the execution of a recursive process and the recovery of earlier instantiations.
Both, the programming paradigms and the software engineering, required modularisation. What appears trivial from the point of view of a functional language becomes more sophisticated at the level of procedural languages where call-by-reference and call-by-value together with random access memory and mixed local and global variable spaces made it very easy to generate memory leakage and overflow as well as incorrect computations caused by programming mistakes or unintendend side-effects.
Having decomposed a huge algorithm into a set of many smaller ones it is quite natural to try and execute those that are mutually independent in parallel. Of course this is possible as soon their variable spaces (or memory access) are disjoint, which requires much discipline in software development (variable access) and algorithm design (side effects).
Hence, the next step was to encapsulate parts of a program into safe and isolated places which then were required to receive input from and send output to other components; i.e. they needed to communicate. Together with multi-threaded processors and time-sharing operating systems with shared and reserved memory per process and thread, things became quite complicated because integrity had to be maintained throughout space (i.e. memory) and time (i.e. concurrent program execution).
Two programming paradigms emerged: object-oriented programming offers a high-level language to the software designer and frees him from thinking about memory management and process interdependencies — at the price of leaving the former to a garbage collector and the latter to a virtual machine with exception handling for fault handling.
The programming paradigm that best fits into the metaphor of cloud computing would be (not surprisingly) a rather old method that was known as blackboard architectures, [24, 14, 13]: whoever needs access to information, reads from a blackboard the data required and writes the result to the blackboard so as to present it to anyone who might need it later. No-one needs to know who provided the required information nor who might rely on the results; all one needs is some agreement on the semantics of the blackboard message language. Should there be not enough information available, a single process would simply wait for it or, in enhanced versions, announce to the blackboard that it requires some information to trigger another process that can provide it. It is basically equivalent to the tuple-space memory model of concurrent programming as, e.g., realized by Linda, [17, 15]. Abstracting from the idea of one local blackboard to a decentralised communication network with information request message broadcasting and delivery the result is what today is known as distributed shared memory (DSM).
Either way, all programming paradigms share one fundamental property. It is that a

program, when executed, initiates and performs a sequence of actions that depend on input data and deliver output data where the correctness of this process with respect to the supplied information relies on the fact that no other process interferes with it.

Chapter 2

Information processing

In this chapter, we generalise from the different levels of details that were presented in Chapter 1. From now on, we will speak of *processes* only and, since such processes are usually described by algorithms written down in a specified language, we call them *programs*.

2.1 Semantics of sequential programs

The semantics of a program basically is, what it does:

1. A program takes some input and delivers a corresponding output; hence its semantics can be described by a relation (usually, a function).

2. A program performs sequences of operations each of which somehow alter internal memory in the process of transforming the input to the output; hence, a log of all internal state changes can be considered one form of the program's semantics.

3. A program starts off with an initial memory (and variable assignments) representing the input and stops (if it stops) on a final memory state where all of its intermediate steps preserve the required specification; hence such a specification is the program's semantics if it is satisfied all time during and after execution.

4. Considering the program itself and a description of the initial memory to be just a complex syntactical expression, this expression can be decomposed systematically until an atomic level or a normal form is reached; hence the definition of the process of deriving this normal form is another form of the program's semantics.

Obviously, semantics is linked to what a program actually does; and a specification is a description of what a program is supposed to do. To avoid failure, we want to avoid that the semantics does not match the specification.

2.1.1 Specification and verification

A specification describes the desired behaviour of a system. It should be complete, i.e. it should cover all situations in which the system is expected to live and all behavioural responses the system can exhibit, and correct, viz. it should exactly describe how to react in a certain situation. Having in mind a rational and goal-directed behaviour one usually implicitly expects a system to behave deterministically which requires a specification to be a functional (more specifically, bijective) description.
Left totality of a specification function requires the ability to describe all possible situations in their entirety, meaning that the model of the world the system lives in is complete and correct, too (cf. the frame problem of artificial intelligence, [49]) and right totality requires equivalence of specification and implementation with respect to the model. Completeness without inconsistencyis, in general, impossible to satisfy which is why it is simply assumed to be true (with the AI's open world assumption versus closed-world-assumption, [74, 23]). To prove the latter one has to verify the program by proving that it behaves exactly as prescribed by the specification. Hence, verification is always relative to the specification rather than the model, let alone the real world.
The language of specification needs to be chosen wisely. Being a logic, we want its structural rules and their interpretations to correspond to the programming language, the compiler or interpreter and the actual execution of the program. Hence, a specification language must be powerful enough to describe the behaviour of the process that is to be specified and its interpretation must comply with all consequences that arise from building complex programs of the chosen programming language, compiler or interpreter and the actual machine it is running on. For as process descriptions (i.e. programs) are elements of programming languages one seeks to give a semantics for the language such that any actual, real process (or program execution) is fully "predictable" (presupposing correctness of compilers and machines) from its abstract specification. Hence, to (provably) avoid glitches, faults and eventual failures, we need a thorough specification to which a program's semantics can be compared. Especially,

1. the more complex a system, the more important it is to specify it and
2. the more complex a system, the more important it is to verify it.

Yet, there are several special properties that may or may not be important to the programs under consideration: Sequentiality requires processes to be composable by executing them one after another; compositionality refers to the principle of defining the meaning of complex expressions by way of the meaning of its components; concurrency means to describe processes in a system running in parallel (in time and/or space; i.e. the environment or memory). Also, different programs behaving equally should have the same meaning—but most specifications do not require there is exactly one program that is correct.

2.1.2 Different flavours of semantics

The meaning of a program can be described in several ways (which, the other way round, also impose requirements onto the specification language). What is common to all kinds of semantics is that a semantics assigns a meaning to an expression (from a language). Hence, the meaning of a variable `X` occurring in a program depends on its value, that is, the (current) content of a memory cell to which the name (or "pointer") `X` refers to, and the meaning of a program is its operating on the meanings of these variables in terms of the syntax of the programming language and the semantics of its complex expressions.

The following three sections provide a short and informal overview illustrating the common characteristics and differences of the "most popular" types of semantics.

Operational semantics

is a rather one-to-one mapping of programming language constructs to term-rewriting rules that are designed in a way to decompose a complex program into one large term consisting of atoms only. It works for ground terms and rule schemes like $\texttt{add}(\texttt{p},\texttt{q}) ::= (\texttt{p}+\texttt{q})$ such that, e.g.

$$
\begin{array}{ll}
 & \texttt{add}(\texttt{p},\texttt{q}) \\
 & \quad |\!:\ \text{Semantics of outer } \texttt{add} \text{ expression} \\
\vdash & (\texttt{p}+\texttt{q}) \\
 & \quad |\!:\ \texttt{p} \text{ is another addition directive} \\
\vdash & (\texttt{add}(\texttt{X},\texttt{Y})+\texttt{q}) \\
 & \quad |\!:\ \text{Semantics of inner } \texttt{add} \text{ expression} \\
\vdash & ((\texttt{X}+\texttt{Y})+\texttt{q}) \\
 & \quad |\!:\ \text{Resolve the variable pointers} \\
\vdash & ((x+y)+\texttt{q}) \\
 & \quad |\!:\ \text{Use the current memory state ...} \\
\vdash & ((1+2)+\texttt{q}) \\
 & \quad |\!:\ \text{... to determine result}
\end{array}
$$

$$
\begin{aligned}
&\quad |\!:\ \text{... to determine result}\\
\vdash\ &(3+\mathtt{q})\\
&\quad |\!:\ \mathtt{q}\ \text{is just a variable}\\
\vdash\ &(3+\mathtt{Z})\\
&\quad |\!:\ \text{Resolve the variable pointer}\\
\vdash\ &(3+z)\\
&\quad |\!:\ \text{Use the current memory state ...}\\
\vdash\ &(3+4)\\
&\quad |\!:\ \text{... to determine result}\\
\vdash\ &7.
\end{aligned}
$$

where $\vdash$ denotes term replacement.
Note that the final result may depend on operator precedence rules or term tree traversal method. Hence, the semantics of $\mathtt{r} := \mathtt{add}(\mathtt{add}(\mathtt{X},\mathtt{Y}),\mathtt{Z})$ is the (partial) function mapping $\{\langle \mathtt{X}, x\rangle, \langle \mathtt{Y}, y\rangle, \langle \mathtt{Z}, z\rangle\} \cup V \mapsto \{\langle \mathtt{r}, x+y+z\rangle\} \cup V'$. Operational semantics calculus can be extended by any arbitrary memory model, too, such that with its explicit management we can also describe much more complex program contructs (one usually adds a stack memory model to model UPN-arithmetic). One could also assume every variable name in a program to represent a fixed memory address and functions $get(m) : \mathrm{Var} \to \mathfrak{D}$ delivering the value stored at an address and $put : (\mathrm{Var} \times \mathfrak{D}) \to M$ where m is one of all possible memory configurations M. For the ease of reading, we simply write X to refer to $get(m)(\mathtt{X})$ and $m[\mathtt{X} \leftarrow\!\shortmid\ Y]$ to show that the memory content for $\mathtt{X}$ has changed to Y: $put(\mathtt{X}, Y)$. Then every rule needs not only to take into account the structural term rewriting but also memory access associated to every rule application:

$$\langle m, \mathtt{add}(\mathtt{X},\mathtt{Y})\rangle \Longrightarrow \langle m[\mathtt{X} \leftarrow\!\shortmid\ X+Y], \mathtt{X}\rangle$$

states that the `add`-command stores the result of the addition of its arguments at the memory address of its first argument and also "returns" this address (i.e. it evaluates to the term stored at the corresponding address). With corresponding rules, one can derive

$$
\begin{aligned}
\langle m, \mathtt{R} := \mathtt{add}(\mathtt{add}(\mathtt{X},\mathtt{Y}),\mathtt{Z})\rangle\ &\vdash\ \langle m[X \leftarrow\!\shortmid\ X+Y], \mathtt{R} := \mathtt{add}(\mathtt{X},\mathtt{Z})\rangle\\
&\vdash\ \langle m[X \leftarrow\!\shortmid\ X+Y, X \leftarrow\!\shortmid\ X+Z], \mathtt{R} := \mathtt{X}\rangle\\
&\vdash\ \langle m[X \leftarrow\!\shortmid\ X+Y, X \leftarrow\!\shortmid\ X+Z, R \leftarrow\!\shortmid\ X], \mathtt{R}\rangle\\
&\vdash\ \langle m[R \leftarrow\!\shortmid\ X+Y+Z], \mathtt{R}\rangle\,.
\end{aligned}
$$

So with appropriate rules defining the semantics of a programming language, we can "trace" a program by stepwise execution of its commands and the continuous updating of the memory. This way, operational semantics corresponds to a state transition system with M being the set of states and the rules defining possible transitions between them.
As an example, consider the program

```
while B ≠ 1 do
        M_R := S_L ;
        M_L := S_R ;
        read(B);
od;
M_R := 0 ;
M_L := 0
```

which shall describe a two-wheeled robot's wandering behaviour: Reading from two light sensors $\mathtt{S_L}$ and $\mathtt{S_R}$ on the left and right front of the robot, their values (representing light intensity) are used to control the motors $\mathtt{M_R}$ and $\mathtt{M_L}$. This control program is supposed to result in a phototactical behaviour of the robot unless its front bumper B reports collision. A (partial) derivation of the program's semantics with $m''(M_R) = m''(M_L) = 0$, $m''(B) = m'(B) = \mathbf{0}$, and $m''(S_{L/R}) = m'(S_{L/R})$ is

$$
\begin{aligned}
& \langle m, \textbf{while } \mathtt{B} \neq \mathtt{1} \textbf{ do } \mathtt{M_R} := \mathtt{S_L}; \mathtt{M_L} := \mathtt{S_R}; \textbf{read}(\mathtt{B}); \textbf{od}; \mathtt{M_R} := \mathtt{0}; \mathtt{M_L} := \mathtt{0}. \rangle \\
& \quad \text{|: Let } C = \mathtt{M_R} := \mathtt{S_L}; \mathtt{M_L} := \mathtt{S_R}; \textbf{read}(\mathtt{B}); \\
\vdash \quad & \langle m, \textbf{if } \mathtt{B} \neq \mathtt{1} \textbf{ then } C; \textbf{while } \mathtt{B} \neq \mathtt{1} \textbf{ do } C \textbf{ else skip fi od}; \mathtt{M_R} := \mathtt{0}; \mathtt{M_L} := \mathtt{0}. \rangle \\
& \quad \text{|: We assume } B \neq 1 \text{ evaluates to } \mathbf{1} \textit{ on } m \\
\vdash \quad & \langle m, C; \textbf{while } \mathtt{B} \neq \mathtt{1} \textbf{ do } C \textbf{ else skip fi od}; \mathtt{M_R} := \mathtt{0}; \mathtt{M_L} := \mathtt{0}. \rangle \\
& \quad \text{|: We assume the execution of } C \text{ leaves us with } m' \text{ where} \\
(*) \quad & \quad \text{|: } B = \mathbf{0} \text{ and } M_L = m(S_R), M_R = m(S_L). \\
\vdash \quad & \langle m', \textbf{while } \mathtt{B} \neq \mathtt{1} \textbf{ do } C \textbf{ else skip fi od}; \mathtt{M_R} := \mathtt{0}; \mathtt{M_L} := \mathtt{0}. \rangle \\
& \quad \text{|: Since } B = \mathbf{0}, \\
\vdash \quad & \langle m'', \textbf{skip} \rangle .
\end{aligned}
$$

The reason we chose this example will become clear later; it is important to understand that this program behaviour crucially depends on the environment specifying the current value of B (see line $(*)$).

Denotational semantics

is a method to describe a program's meaning by grounding its syntactical elements and structures in mathematical functions which, atthis abstract level could be composed so as to construct the meaning of a complex program in terms of compositions of denotations, [87] and [81, 82]. Such a semantics requires some extra effort in comparison to operational semantics: Domains and methods to access them. The most important one is the *store* σ (or memory) which is just a vector of n arguments each of which represents a memory cell holding a natural number ($\sigma = \mathbb{N}_0{}^n$). Σ is the countable set of all stores; i.e. the set of all memory configurations. *get* delivers the value for a variable $\mathtt{X}_i$ by picking the i-th

component from σ ($get(e)(\mathtt{X}_i, \sigma) := \pi(e(i))(\sigma)$) and $put(e)(\mathtt{X}_i, v, \sigma) = \sigma'$ defines the store resulting from updating the i-th component of σ by the value $v \in \mathbb{N}_0$. Both functions are parametrized by *environments* e which allow different interpretations of a variable name (i.e. one and the same variable name $\mathtt{X}_i$ may refer to *different* storage cells depending on the environment e). This allows binding variables to different values which means that with regard to the context (i.e. environment) a variable's value may vary by the different memory cells the pointers refer to in different environments.
As an example, we demonstrate the idea behind this approach along the same example as above (again, we skip the details including the different interpretation functions for expressions and commands or even the least fixed point interpretation of a terminating loop).
The syntactical operation of term replacement "$\vdash$" is now replaced by equality of interpretations $\lfloor \mathtt{p} \rfloor^e_\sigma$ which denote the sets of environments e and variable bindings σ that are valid models of the program.
Note that the value of $\mathtt{B}$ is not determined by a **read** operation that delivers the value $get(m)(\mathtt{B})$ as in operational semantics. Here, the environment e determines the set of satisfying assignments $\lfloor\, \mathtt{B} \,\rfloor^e_\sigma = \pi(e(b))(\sigma)$ where we assume $\mathtt{B}$ to be stored in the b-th component of σ:

$$
\begin{aligned}
& \lfloor \textbf{while } \mathtt{B} \neq \mathtt{1} \textbf{ do } \mathtt{M_R} := \mathtt{S_L}; \mathtt{M_L} := \mathtt{S_R}; \textbf{od}; \mathtt{M_R} := \mathtt{0}; \mathtt{M_L} := \mathtt{0} \rfloor^e_\sigma \\
& \quad \text{|: We assume the bumper does not report a collision: } \lfloor \mathtt{B} \rfloor^e_\sigma = \mathbf{0} \\
= \; & \lfloor \mathtt{M_R} := \mathtt{S_L}; \mathtt{M_L} := \mathtt{S_R}; \textbf{while } \mathtt{B} \neq \mathtt{1} \textbf{ do } \mathtt{M_R} := \mathtt{S_L}; \mathtt{M_L} := \mathtt{S_R}; \textbf{od}; \mathtt{M_R} := \mathtt{0}; \mathtt{M_L} := \mathtt{0} \rfloor^e_\sigma \\
& \quad \text{|: After assignments, } \mathtt{M_R} \text{ and } \mathtt{M_L} \text{ have the values of } \mathtt{S_L} \text{ and } \mathtt{S_R} \\
= \; & \lfloor \textbf{while } \mathtt{B} \neq \mathtt{1} \textbf{ do } \mathtt{M_R} := \mathtt{S_L}; \mathtt{M_L} := \mathtt{S_R}; \textbf{od}; \mathtt{M_R} := \mathtt{0}; \mathtt{M_L} := \mathtt{0} \rfloor^{e'}_{\sigma'} \\
& \quad \text{|: We assume the bumper reports a collision: } \lfloor \mathtt{B} \rfloor^{e'}_{\sigma'} = \mathbf{1} \\
= \; & \lfloor \mathtt{M_R} := \mathtt{0}; \mathtt{M_L} := \mathtt{0} \rfloor^{e'}_{\sigma'} \\
& \quad \text{|: After assignments, } \mathtt{M_R} \text{ and } \mathtt{M_L} \text{ have the values } \; 0 \\
= \; & \lfloor \mathtt{skip} \rfloor^{e''}_{\sigma''}
\end{aligned}
$$

which leaves us with a full stop in e''. Since

$$\pi(e'(b)(\sigma')) = \mathbf{1},$$

the following assignment sets both motor values to 0:

$$\pi(e'(l)(\sigma'')) = \pi(e'(l)(\sigma'[\mathtt{M_L} \hookleftarrow 0])) = 0$$

and the same for r and $\mathtt{M_R}$, respectively.
Note that once $\mathtt{B}$ delivered the value $\mathbf{1}$ *at the time of* evaluating the loop condition, the program will always bring the robot to a complete halt and then terminate. Even if *during* the assignments of 0 to $\mathtt{M_L}$ and $\mathtt{M_R}$ the environment e' is changed to e'' with

$$\pi(e''(b)(\sigma')) = \mathbf{0},$$

the robot won't start moving again even if the bumper contact is released.
On the other hand, e demonstrates the problem of concurrent memory access: If there was *another* program, it might have caused a change on e'' resulting in different versions of σ'' at, say, l. Then, the signal to the left motor is determined by the environment or conflicting program rather than by our program (forcing the left motor to stop).
This example appears only to be a syntactical variant of operational semantics, but here we have the "history" of all variable values given by the set of all environments and, much more important, the value of `B` depends on the environment rather than some expected interaction by **read**. Therefore, any other program working on the same memory cells (and not the cells that variable names point to) may alter (if not even control) a program's behaviour.

Axiomatic semantics

finally gives us a tool to derive a correct program along a first order logic specification of its desired behaviour and a method to verify that a given program actually does what it is supposed to do. The core idea are so-called Hoare-triples $\{\!|\, Pre \,|\!\}\, C \,\{\!|\, Post \,|\!\}$ where *Pre* and *Post* represent conjunctions of first order logic formulae with equality describing the pre- and postconditions of executing a command C. We describe the method using our our example code again. The arabic-numbered lines contain the program code and the alphabetically labelled lines are "comments" containing the required pre- and postconditions that hold between the program instructions.
Note that this formalisation does not require states σ or functions e on them. Program variables like `B` are interpreted by variables B and free variables are treated as universally quantified (i.e. $B = X$ means that *whenever* the program runs across this requirement, `B` must evaluate to the same value as `X`).
The initial setting for our program shall be that the bumper is released, the sensors have a zero reading, and the motors are stopped. We also require for this version of the control program that the values for sensors and motors always add up to integer multiples of 100:

$$\begin{array}{rclrcl} \varphi & \equiv & M_R = S_L & \psi & \equiv & M_L = S_R \\ \chi_M & \equiv & (M_R + M_L)(100) = 0 & \chi_S & \equiv & (S_R + S_L)(100) = 0 \end{array}$$

To emphasize the difference to the previous methods of specifying a semantics, we here use a function **poll** to read a variable's value. It *locks* the variable's memory cell for write access until it is explicitly **release**d.

a		$\{\!\|\, B = S_L = S_R = M_L = M_R = 0, \varphi, \psi, \chi_S, \chi_M \,|\!\}$
1	**while** `B` $\neq 1$ **do**	
b		$\{\!\|\, B \neq 1, \chi_M \,|\!\}$
2	**poll** $\mathtt{S_R}, \mathtt{S_L}$	$\|$: Variable update
c		$\{\!\|\, \chi_S, \chi_M \,|\!\}$

3 $\quad\quad \mathtt{M_R} := \mathtt{S_L}$; $\{\!|\, \varphi \,|\!\}$ **release** $\mathtt{S_L}$;
4 $\quad\quad \mathtt{M_L} := \mathtt{S_R}$; $\{\!|\, \psi \,|\!\}$ **release** $\mathtt{S_R}$;
e $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \{\!|\, \chi_M \,|\!\}$
5 $\quad$ **od**
f $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \{\!|\, B = 1, \chi_M \,|\!\}$
6 $\quad \mathtt{M_R} := 0$;
g $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \{\!|\, B = 1 \,|\!\}$
7 $\quad \mathtt{M_L} := 0$;
h $\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \{\!|\, B = 1, \chi_M, \xi \,|\!\}$

where $\xi \equiv M_R = M_L = 0$.
Obviously, (a) holds due to the memory initialisation. Also, (b) is true because the loop body is entered only if $B \neq 1$. For the same reason $B = 1$ holds in (f)-(h) and χ_M, ξ are true in (h) because $M_R = M_L = 0 \equiv \xi$ after execution (6) and (7) and $(0 + 0)(100) = 0 \equiv \chi_M$.
The loop body is a bit more tricky: We assume (the reason will become clear later) that $M_{L/R}$ can only be changed by the commands in (3)-(4) and (6)-(7). $S_{L/R}$ and B can be overwritten by some other process, i.e. can change their values at any time except for while executing the *poll* command in (2): it returns the current and possibly altered values for $S_{L/R}$ and B but guarantees that S_L and S_R together sum up to 100 and "locks" the variables against other process activities unless they are released.
Let us suppose, that χ_M is true when entering the loop. Then it stays true until (3) where M_R is assigned a new value but M_L still carries the old one. But since polling in (2) ensures that χ_S holds true, χ_M is true after assigning the new value of S_R to M_L. Note that φ and ψ are true only at the very beginning and in lines (3) and (4) between assignment and release. Hence, χ_M is true before and after a loop execution, hence it is an invariant.
Skipping (6)-(7) we find that at the very end we have $M_R = M_L = 0$, hence χ_M and ξ are true (note also, that we had lost χ_M between (6) and (7).
So while the program is running, the values of $M_{L/R}$ may change constantly but, as a result of the characteristics of $S_{L/R}$ (namely χ_S) also satisfy χ_M most of the time. The program comes to an end only if at some point somehow the value of B changes to 1 and then the program always terminates with $M_{L/R} = 0$ no matter what the values of $S_{L/R}$ are.
Obviously, after termination, our memory holds the following values for our variables: $B = 1, M_L = 0, M_R = 0$. The first one is due to the fact that the loop condition must have failed (otherwise the program would not have come to a halt); and the latter ones are result of the two assignments at the end of the program. By convention, the memory is initialised with all cells set to 0, which is why at the beginning $T = 0$. Hence, we could state that

$$\{\!|\, T = 0 \,|\!\}\ \textbf{while}\ T \neq 1\ \textbf{do}\ C\ \textbf{od}\ ; D\ \{\!|\, T = 1, M_L = 0, M_R = 0 \,|\!\}$$

where T is the loop condition, C the loop body, and D the block of assignments at the

end. The values of $M_{L/R}$ are unspecified before execution of D such that we could write

$$\{\!|\, T = 0 \,|\!\}\ \textbf{while}\ T \neq 1\ \textbf{do}\ C\ \textbf{od}\ \ \{\!|\, T = 1 \,|\!\}.$$

Imagine now that S_L and S_R are two variables that are affected by some external process but that their values always add up to 100 whenever their values are asked for during execution of commands. Let us further assume that S_L and S_R cannot be altered while executing the loop body (this is a very strong assumption; especially in combination with the possibility of external processes changing their values). At the same time, we want M_L and M_R to deliver 100 as a sum.
This can be formulated by

$$\chi := S_L + S_R(100) = 0 \text{ and } \ \xi := M_L + M_R(100) = 0$$

Due to memory initialisation, we have that $\varphi_0 := B = S_R = M_L = 0$ and $\psi_0 := B = S_L = M_R = 0$ are true.
A weakened condition is that, upon first loop entry, $\varphi := B = 0 \wedge S_R = M_L$ and $\psi := B = 0 \wedge S_L = M_R$ are true and, therefore, $B = 0 \wedge \varphi_0 \Longrightarrow \varphi$ and $B = 0 \wedge \psi_0 \Longrightarrow \psi$.
Hence, $\varphi \wedge \psi$ also holds after each execution of C (but not neccessarily *within* C) such that

$$\{\!|\, B = 0, \varphi \wedge \psi \,|\!\}\ \textbf{while}\ T \neq 1\ \textbf{do}\ C\ \textbf{od}\ \ \{\!|\, B = 1, \varphi \wedge \psi \,|\!\}.$$

Note that validity of $\varphi \wedge \psi$ in the precondition is because of initialisation whereas in the postcondition it is because of the two assignments.

These are just simple examples for three popular types of semantics to motivate the following: The crucial thing behind program behaviour are *events* that *change* the *environment*. Every such event is the result of some process computing some value it communicates by publishing the result in a variable. Taking into account a number of processes with concurrent access to memory, we have deduced the metaphor of communicating processes (including all the problems by interference).
Also, we have seen that the semantics of a program is required for and can be used for in two general directions: we can prove or disprove that a program behaves well w.r.t. a specification and we can use a specification to derive a program from it.

A nice, informal comparison of the three different types of semantics in the context of non-deterministic programs is given in [28].

2.2 Assemblies of processes and concurrency

Following the observation described in the introduction, an embodiment of a process would be a physical entity that receives (sensory) input from the environment and acts on it, leaving behind an environment that has changed due to the effects of his acting. Such an

entity is called an agent. According to this definition, agents can be virtually anything in our real world that somehow interfere with the environment. With their (common) environment being the input space they also deliver output by acting: Simple machines, software agents, robots, any kind of functional component involved in a complex system such as an assembly line or operating system.
This approach to analysing processes can be applied to nearly arbitrary levels of detail: An entire production process can be decomposed into business and design processes, software design and implementation, then running different programs (each of which again may consist of several such modules), on different computers (also built from modules) controlling robots with modules for sensory input processing etc.

2.2.1 Behaviour by design vs. behaviour by collaboration

Distributed artificial intelligence and multi-agent systems offer a similar interpretation of interacting and cooperating individual agents that together attain a common goal or exhibit a desired behaviour, [90].
The "creationist" approach is to thoroughly specify, design and carefully implement the individual components to deterministically perform the desired actions in appropriate situations. This also requires to design the entire assemblage of cooperating agents in order to guarantee deterministic behaviour. Complex systems cannot be formally specified to a sufficient extent; and even if they were, the verification would require far too much effort. Furthermore, a true verification still cannot avoid nondeterministic behavior: In reality, something always goes wrong. Hence, the second ("evolutionary") approach is to build primitive modules (without too much functionality and, hence, semantics and conditions) which together by their interaction act as one large process (with a complex functionality).

The creationist approach

is in principle able to deliver systems that behave exactly in the same way they are supposed to do; both on microscopic as well as a macroscopic level — and both with respect to the functional correctness of each individual component and their pairwise interaction. Its drawback is that specification is tedious and development hardly ever correct w.r.t. specifications, so that the actual behaviour is not guaranteed to be the same as intended by the specification (especially in non-isolated environments that impose a certain degree of nondeterminism on the system components). Correcting ("debugging") a system, its components and communication methods is a task that comes close to an entirely new system design which may resolve previously encountered problems but may be fallible just as before — only for perhaps different reasons.

The evolutionary approach

on the other hand is a process that constantly refines a system of simple, atomic modules or processes. Assuming some kind of monotone and continous change a constant refinement of the system components may allow maintaining at least a certain degree of reliability (modulo short lags during readjustments). The fundamental disadvantage is that "twiddling with black boxes" usually happens undocumentedly, i.e., the system and its components may change over time whereas its specification doesn't (or even does not exist at all). Also, with changing functionality of modules one might miss to adapt communication, too (resulting in buffer overflows, type inconsistencies etc).
Finally, all more or less discrete components only allow tweaking up to a certain degree at which a turning point is reached: Then, a system breakdown is also nearly impossible to fix without entire reimplementation — but for a different reason: Due to the constant refinement during runtime without any documentation, the actual functional meaning of isolated modules and, let alone, components of modules, has become totally unclear.
Creationism is bound to fail due to nondeterminism of the environment whereas evolution is bound to fail due to determinisim of specification.

2.2.2 Agents

Agents are one of the most recent metaphors for goal-directed (i.e. rational, [1]) processes that interact with their environment, [91, 90]. While agents of these kind are themselves already complex software systems, simplest processes and machines (i.e. "modules") can be thought of as being agents, too: Small and simple electronic or mechanical parts comprising a complex machine, small and simple (cognitive, here: rational) processes that together form "intelligent" behavior and simple computational units that by suitable connections allow universal function approximation, [41, 6, 48, 78].
Assemblies of agents together may or may not follow an individual or general plan. They act in a shared environment and they usually perform a collaborative effort—whatever that is (even if not intended or seemingly "counterproductive").
The resulting system behaviour is then predetermined or at least close to the intention ([1, 7, 72]) or freely emergent [10, 12]. Hence, an entire set of modules with its collaborative performance can be thought of being another, atomic module at a higher level of abstraction.
With the environment being a collection of information and agents accessing, transforming and aggregating information, blackboards, developed in the early 80's [13, 14, 24] and distributed shared memory models such as in Linda (from the mid-80's, [17, 15]) have now gained new interest as a memory model under the name of tuple spaces, [77], in object or service oriented models.
The fundamental problem that we come across when dealing with increasingly complex software or hardware systems nowadays is:

1. The more complex a system, the more difficult it is to specify it.

2. The more complex a system, the more difficult it is to verify it.

Therefore, increasing need for specification and verification comes with increasing difficulties; hence there is a strong need for *simple models* and *simple logics* in order to overcome these difficulties.

2.2.3 Programs without semantics

Brooks's paper entitled "Intelligence Without Reason", [12], tries to give a belated vindication of his earlier work on "Intelligence without representation" by a constructivist approach to intelligent behaviour.
On the one hand, the recent rediscovery of problem solving by emerging behaviour of collaborating agents advocates the approach of cooperating and independently interacting modules. On the other hand it is a fact that most systems nowadays are *missing* the "reason" in the sense that the knowledge the developers have put into the design has been lost for several reasons (the system requirements were never specified at all, the specification has been lost, the system has been constantly "refined" without adapting the specification or verifying the changes, see section 2.2). In other words, the intention of the programmer cannot be recovered from the representation.
Most systems that *fail* need to be analysed "post mortem". Analysis is nearly impossible, if there is no specification. But in most cases the failure is due to a weak specification and, hence, the system does *something* similar to what one wants, but it is missing a proper semantics. It is, so to say, a "*program without semantics*".
Motivated by Brooks's subsumption architecture and experience from cognitive robotics experiments, we developed a more abstract model called "asynchronous complex module environments" [61]. This model is able to describe nearly any kind of processes that somehow interact in a common environment. The goal is to *induce* hypotheses about what the entire system is meant to do: From observing what it does and from its general architecture we want to infer hypotheses about a suitable specification.
This hypothesis can be checked for obvious errors or statements that evidently contradict assertions about the desired program behaviour. And this way, one might be able to find the reason for the system failure more efficiently.

Chapter 3

Embodiments of processes

This chapter describes previous work that, in its origin, was focused on a real-world implementation framework for multi-agent systems. With all agents arranged in an assembly line, the set of agents forms a sequentially composed system of processes. When breaking up the sequential design or taking out a global clock timer, some components may act independently from others whereas others (mutually) depend directly or indirectly on each other. Such a set of task-cooperative or resource-competing agents is a system of concurrent processes.

3.1 ACME: Autonomous Complex Module Structures

ACMEs, see [61], resulted from the need for a formal framework to describe cognitive agent and cognitive robot environments. In the course of the PARIA (Platform for Autonomous Robots and Intelligent Agents) project, [60, 26, 37] running from 2005-2008 at the University of Augsburg and from 2009-2010 at the University of Applied Sciences Bonn-Rhein-Sieg, we developed an integrated framework for both virtual agents and real robots that cooperate and compete in a common environment. The original setting was to enable Lego-RCX robots to compete in a Robo-Rally-like game. Over several generations of continous improvements we finally ended up with embedded Linux controlled robots communicating via WLAN and exchanging information using a blackboard server. Using the benefits of TCP/IP communication, Linux device file systems and multi-threaded processes, one could easily implement arbitrary agents that act locally, or design systems of agents (or rther modules) in a Brooks-like manner, [38, 83, 22]. In other words, an increasingly robust framework for embodied agents (i.e. robots running processes) naturally led to an increasingly abstract description of process communication.

3.1.1 The general idea behind ACME

The last section concluded with the description of a program's semantics in terms of events, messages (communication) and environments. The same holds true for "real life" processes that interact within a common environment. Therefore, at a certain level of abstraction we can identify the following classes and the equivalences between their different interpretations according to their field of application:

- The I/O-behaviour of a program, required resources and produced results by a machine, sensory data and actions of a robot, read/write access to memories, etc.
- Interface variables of a program/procedure as a part of the global memory, messages on a blackboard, global (external) events changing the environment in which a robot acts, etc.
- Processes executing programs, robots acting according to their models (and goals), processors running processes, etc.
- State transition graphs for simple automata, logical descriptions of world models, plans and actions for robots and agents, functions for denotational specifications, etc.
- Behaviour of agents and robots, traces of events, state sequences generated by transitions, snapshots by memory maps, etc.

Since we shall use ACMEs to describe the abstractions of the different readings, we will from now on use the following terminology that, depending on the scenario we work in, can be translated into the corresponding idiosyncratic notation.

- A *specification* is a high-order description of what a process shall do. It requires a specification language with a well defined semantics.
- A *program* is an algorithm defined in a specific programming language. Its purpose is to give a recipe of how to compute a function.
- A *module* is an entity that takes a program and executes it in a *(global) environment* from which it takes information and into which it infuses information. A module usually is equipped with some "inner life" which, according to it, is part of the environment, too. Since it is invisible to others,we shall call it *local* or *interior environment* (since there is no such word as invironment).
- A *system* is a set of *modules* sharing a common environment.
- A *snapshot* is an instantaneous image of the environment together with all its modules and their states at a certain point in time.

- A *process* is a sequence of *events* where each event is any change in the environment. Sets of events may form *complex events*.
- A *trace* is a sequence of footprints of a process, it is a (ordered) set of snapshots.
- If an event can occur only if the snapshot satisfies certain properties, then the process *depends* (on the environment).
- If a certain snapshot is a result of an event, we say that the event *brings about* (a certain environment).

3.1.2 Observing a system by collecting snapshots

Observation over a period of time means to collect data produced by the system's states as well as how data changes over time and during the system's behaviour. Hence, the system behaviour determines the structure of collected data—but given data, we need not neccessarily be able to draw valid conclusions about the processes, let alone the involved programs or even the specification.

Examples

A one-instruction two-variable process. Let us a assume the simplest case first: The system S consists of only one module m running a program p that implements an algorithm to compute a total function $f : D \to D$. The argument and result are stored in separate variables; the variable names in the program are mere pointers to memory in the (global) environment. Let us also assume that m performs one execution of p. Every execution of p creates one process P which becomes visible as the one and only event in a trace. For $f(x) = x + 1$ (implemented as $p = \mathtt{y} := \mathtt{x} + \mathtt{1}$) we think of P as being the process of computing the successor of x and storing it in y. A trace depends on the range of the snapshot and the actual environment. So if the environment only provides two memory cells, one for x (called "x") and one for y ("y"), and the former holds the value 4, then the latter will have the value 5 after the execution. The trace in this configuration would be

$$[\{\langle \mathtt{x}, 4\rangle, \langle \mathtt{y}, \_\rangle\}, \{\langle \mathtt{x}, 4\rangle, \langle \mathtt{y}, 5\rangle\}].$$

Note that y is "known" in the first step of the trace already since we assume the snapshot range to be constant w.r.t. the variable names and since we assumed the range to cover the entire memory of only two cells (named x and y).[1] With many such traces, one might discover that *every* event generated by *this process* leaves us with an environment $E \subseteq \{\langle \mathtt{x}, x\rangle, \langle \mathtt{y}, x + 1\rangle\}$. Since E may provide many more memory cells than just the ones

[1] For later, it is worth mentioning that x denotes one and the same memory cell all the time. It can be considered a physical memory address but not a pointer. Hence we can only implement call-by-value programs on global variables.

P works on, it is quite a challenge to identify x and y as being "characteristic" for P. But if we discovered such a dependency (see, e.g. [62]), we could also analyse a trace

$$[E_0, \ldots, E_i, E_{i+1}, \ldots, E_{n-1}]$$

with many values $i \in \mathbf{n}$ such that

$$\langle \mathtt{x}, x \rangle \in E_i \text{ and } \langle \mathtt{y}, x+1 \rangle$$

and identify P as the process that adds 1 to x. This way, one would have induced the semantics of p from the trace it leaves when executing P.

Control loops, sensors and actuators. Looking at agents or robots (or in general any program acting on arbitrary input and output variable spaces), we need to add special modules to poll or publish values. For reasons of simplicity, we assume all variables to be of the same type; that is, all of its possible values to be from the same domain D.
First, we add modules acting as sensors. Sensors collect input from the unobservable (external and internal) environment such that the "sensing function" can be identified with its value $s :\to D$. The system also contains a set of modules $m_j : D \to D$ where each module implements a function $f_i : D \to D$ by a program p_i. This definition can easily be extended to modules, programs and functions which take several arguments and deliver several output values (we restrict ourselves to unary endofunctions only for reason of a simpler notation). Note that here every module may have an individual signature and, of course, domains and codomains can also consist of more complex structures (products to model n-ary functions). Examples for more complex modules are logic gates `and` : $\mathbf{2} \times \mathbf{2} \to \mathbf{2}$, vending machines `m` : $\mathit{Money} \times \mathit{ProductCode} \to \mathit{PurchasedItem} \times \mathit{Change}$, simple control circuits (see below), or an industrial robot assembling several parts into a product. Obviously, sensors are assumed *independent* in the sense that they simply take information from the outside and provide them to the system, whereas all other modules depend on existing input values of the right type and within the specified range. Taking a more general view, we can distinguish three special types of modules that depend on the signature of the implemented functions. The normal case is a function $f : D \to C$ where D and C are products of domains (types). If $D = \emptyset$, f simply provides values from C. If $C = \emptyset$, then the module implementing f has no influence on any other modules. We call them initial and terminal modules and in terms of robotics, we could also call them sensors and actuators. Then, indeed, they do not affect other modules *directly*, but they do affect the environment through side-effects and, hence, have an *indirect* influence on the behaviour of other modules (and the entire system).
We now consider the case of a robot implementing a phototactical behaviour as in Chapter 2 again; but this time from the point of view of multi-agent system design: Two initial modules s_L and s_R provide integer values representing the light intensity recorded by two photo diodes on the left and right side of a robot's body. A third initial module, b,

holds the value of a binary bumper switch that is **1** if the robot's front is in contact with an obstacle and **0** else. Two terminal modules m_L and m_R translate incoming integer values into electric current driving the left and right wheels. In between there is a module implementing a function *drive*, that simply redirects output from s_L and s_R to m_R and m_L, resp. If the bumber b signals contact it stops the entire robot by setting the input values of the terminal modules to 0 whatever the inputs from the initial modules are. For example, one could specify *drive* as as a function on 8-bit integers and a Boolean bumper signal as

$$\texttt{drive} : 2^8 \times 2 \times 2^8 \to 2^8 \times 2^8 \text{ with } \texttt{drive}(R, B, L) = \langle L, R \rangle$$

which would result in the desired phototaxis behaviour of a two-wheeled robot. The results are interesting; especially with many such robots sharing one common (real world) environment: Should there be just one light source in an otherwise dark room, the robots will gather around around this light source; should every robot carry its own light (which itself is unable to see), the robots will tend to follow each other in a meandering procession. Note that in this case the environment is, in fact a non-deterministic real-world environment rather than a closed system of processes with concurrent access to memory. This means that we may encounter *unpredictable* (and unwanted) interference between processes, especially when this environment is also inhabited by additional processes that are not part of our specification.

An industrial robot can assemble products only if it receives all necessary parts; if properly programmed it won't start assembling the parts if there is one missing but it could fail, if some parts are of the wrong type. With all parts available in an admissible version, the final result of assembly may differ depending on the input (e.g. differently coloured parts). Also, more "intelligent" robots might be able to combine several programs and execute different assembly steps depending on different types of parts. The entire production process might fail with "rogue" robots interfering with the otherwise correctly behaving robots.

So far, we have modelled abstract modules that perform a certain task individually as a simple "black box". But the most important thing is to link them together such that one produces the input for another.

3.2 Relational models of modules and channels

Using a *channel* (or, simply, an *arrow*) we connect a module's output component to a modules input where (in allusion to network protocols) the different components of the function signatures are called *ports*. Since

- several modules can connect to one and the same destination port,
- every module has its own runtime behaviour,
- due to channel congestion or lags,

- modules may receive no, not sufficient or inconsistent input data,

the entire system's behaviour is far more complex (and far more difficult to specify) than just the set of its components. Just as in verification, *communication* is what makes the difference and causes most problems (also in real life).

Brooks's layered architectures ([10, 11]) consisted of sets of computable functions $f : (2^8)^n \to 2^8$ called *modules* and (branching) channels to pass one function's value to other functions. Higher level-layers included channels that allowed for blocking or overwriting channel communication at lower levels and, in the topmost layer, channels connecting functions and channels, thus modifying the behaviour of channels (i.e. message passing). With ACMEs, this concept was taken to a further level of abstraction where the architectures are described relationally. The strict level architecture has been weakened to a "flat" structure with all different types of channels implemented by modules. Since Brooks's architectures were designed for implementing systems with minimal (hardware) requirements in behaviouristic environments only, ACMEs also include a more sophisticated method for storing and recording variable/value changes over time. This enables one to record "traces" as sequences of system states that are proven means for describing the semantics of a system (see section 2.1.2) and also deliver factual data from which one can induce logic descriptions of a system (see, e.g., [58]).

3.2.1 Modules

Let there be a finite set S of total, computable functions f_i each of which takes $m_i \geq 0$ arguments from possibly different sets and delivers as a result $n_i \geq 1$ values (also, from possibly different sets). Each function

$$f_i : I_1 \times \cdots \times I_{m_i} \to O_1 \times \cdots \times O_{n_i} \tag{3.1}$$

is implemented by a *program* and runs as its own *process*. We assume all I_{k_i}, O_{l_j} to be well-defined subsets of our domain. We can also depict f_i as a "black box":

$$\begin{array}{|l|c|l|} \hline I_1 & & O_1 \\ \vdots & f_i & \vdots \\ I_{m_i} & & O_{n_i} \\ \hline \end{array} \tag{3.2}$$

Two functions $in(f)$ and $out(f)$ deliver domain and codomain for single functions f; for sets of functions, $in(S)$ and $out(S)$ deliver the disjoint union of $in(f)$ and $out(f)$ for all $f \in S$. Note that the values of *in* and *out* are actually products or sets of products. The observable ("external") memory space a set of memory cells, one for each element in $in(S) \dot\cup out(S)$. The state space is determined by all the possible values and value combinations; a single state is

$$\sigma \in \bigtimes_{f \in S} in(f) \times out(f).$$

Verifying such a system by checking all possible configurations is clearly infeasible.

3.2.2 Channels

To "connect" two different modules, we simply draw an arrow from one module's out-ports to another module's in-ports such that for two unary functions f and g their composition $g(f(x))$ would become $x \longrightarrow \boxed{\circ\ f\ \bullet} \longrightarrow \boxed{\circ\ g\ \bullet}$. This connection represents a *sequential* information processing procedure; we could also write $f; g$ to indicate that first f is carried out and then g where the information passed along the arrow is stored and accessed by global variables (see section 2.1.2). Note that the arrow notation does not explain how the value of $f(x)$ is passed to g; nor do we take into account anything about the runtime behaviour of the modules implementing f and g.

As an example we again construct a system that shall exhibit a phototaxis behaviour (see figure 3.1). As already described in earlier sections, this system consists of two sensors

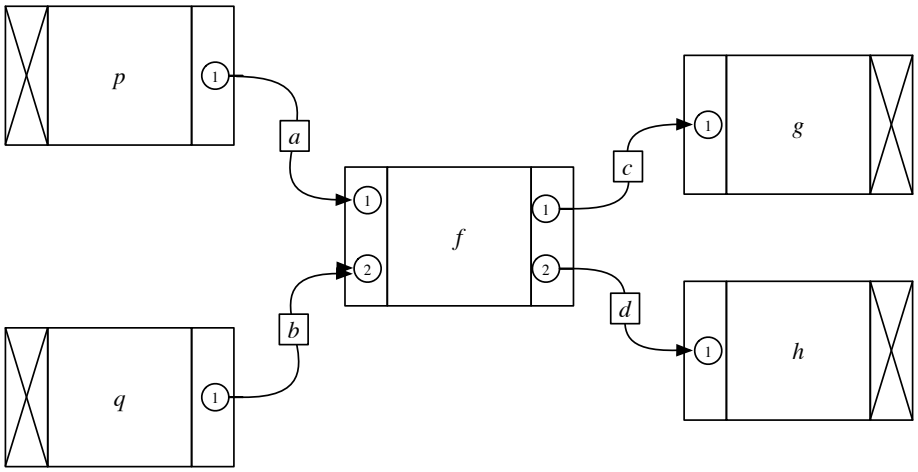


Figure 3.1: Phototaxis by ACMEs

(initial modules), two actuators (terminal modules, motors) and a computational unit that simply maps the input value from the left sensor (q) to the right motor (g).
This structure can be interpreted as a graph with channels a, b, c, d being the edges and all *ports* being the vertices. We write the i-th inport of f as $\langle i|f$ and the j-th outport of g as $|j\rangle g$.
Hence, the input to the right motor g is determined by

$$\langle 1|g = |1\rangle f = f(\langle 1|f, \langle 2|f) = f(|1\rangle p, |1\rangle q).$$

If we understand this equation as equality between values, it means that channels themselves also model equalities. This is, of course, *not* true when taking into account runtime and signal travel time. As a result, the structural description of such a system system

would be false most of the time. This, again, motivates that each channel actually represents a single variable (or, rather, memory cell) and ports as pointers thereto. In Figure

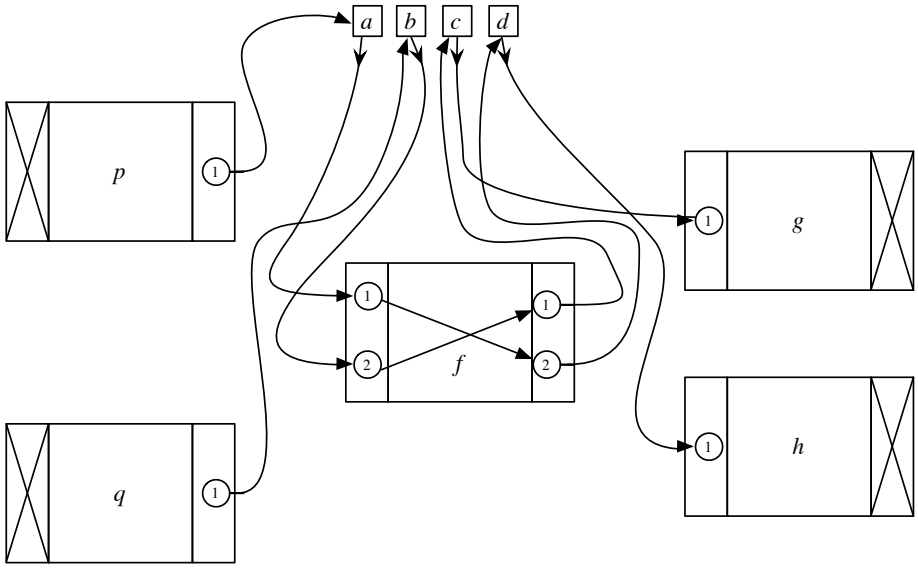


Figure 3.2: A memory based model

3.2, $|1\rangle p \to \boxed{a} \rightarrowtail \langle 1|f$ means that both $|1\rangle p$ and $\langle 1|f$ are pointers to a memory cell with address a. This figure also shows the internal structure of f: $|1\rangle f := \langle 2|f$ and $|2\rangle f := \langle f|1$. Substituting these assignments in the equation above, we have

$$g = \langle 1|g = [\![c]\!] = \langle 2|f = |1\rangle f = [\![b]\!] = |1\rangle q = q$$

and the analogue for p and h (where $[\![a]\!]$ denotes the *content* of the memory cell a). The problem here is that a memory cell can hold only one value at a time which means that at every time, the value of the left sensor q is the same as the value of the right motor g. This is, of course, *not* the case: The sensor p may update a at a higher frequency than f can compute the values of c and d. Also, while it is computing an output, the motor g is running at a *previously* set speed. Or, to put it into a simple phrase: Were the equation correct, the right sensor's signal would always be the same as the left motor's input voltage which is physically impossible on a practical level (by clock timing) and theoretical level (because of signal travel time). Considering a concurrent interpretation, the problem of (in-) consistency becomes even clearer.

3.2.3 Modules at work: Concurrency

The next problem is the following: Imagine f currently computes the values for an input $a = 12$ and $b = 33$. The result is $c = 33$ and $d = 12$. But while f was computing the

output, p updated a with a new value; say 17, such that the memory configuration at this point in time would suggest that $f(33, 17) = \langle 12, 33\rangle$—which is wrong (even though the implementation of all components might be perfectly correct). The next problem is that of message transmission timing: We assume a channel to be any kind of connection, be it a memory cell, a pointer (i.e. a variable) or even a TCP/IP channel. It is clear, that messages need some time to travel and (as evident in the case of TCP/IP) the information packets may be delayed due to any kind of reason or get queued before a bottleneck and then flushed at a higher frequency than the sender originally had sent them (which might result in the receiver skipping many packets). One extreme case would be where p measures a value a_1, its last measurement output was a_2, a_3 is "in the queue", a_4 is the current input value for f, the value f is just working with is x, f's last output was d_1, d_2 is in the queue, d_3 is the current input to h and h is currently running at d_4. If we can describe the system by a sequential process with call-by-value procedures and global variables only, this simple model would be sufficient but, as we have seen, it is not sufficient for more advanced programs or concurrent systems.
Another problem is that, adding the third bumper sensor to the system, three different implementations of the entire system come to mind all of which seem to act equally: The emergency stop signal from r can be duplicated and sent to both inputs quasi-simultaneously, they can be sent to the motor actuators directly by circumventing f or we could add a third input to f. The former versions both suffer from synchronisation problems (motors might stop with some differential time delay), the second one also from the fact that a direct stop signal with fast travel time might be overwritten by f afterwards (resulting in a short stop followed by new accelaration due to light sensor information dating back from before perception of a bumper signal), and the third one from a delayed stop due to additional computation time by f (see figure 3.3). A first simplification is based on the fact that we cannot distinguish between the current environment and our perception. Hence, we would disregard the possible difference between a_1 and a_2. Similarly, we shall only take into account the actual change we perform but not the last one (since we chose the action based on our perception of the environment)[2], such that we assume d_3 to coincide with d_4.
To resolve these problems (channel delay and local inconsistency), [61] already suggested a *buffer model* where every port has two memory cells associated: One that continously receives new input values and a "shadow" copy holding the value of the port when the function was called/completed the last time. This way, any memory snaphot of shadow copies always consists of variable assignments that gurantee local consistency: Every inport consists of two memory areas: One is an "anytime" write memory cell and the second one is a discrete time poll copy of it. We write $\langle i|f$ for the anytime part of an inport and $[i|f$ for its local copy; for outports we write $|j\rangle f$ and $|j]f$. All anytime in- or outports are globally accessible addresses and together form an *interface*.
The behaviour of f is as follows:

[2] A history-sensitive behaviour can be realised by (delay) feedback channels.

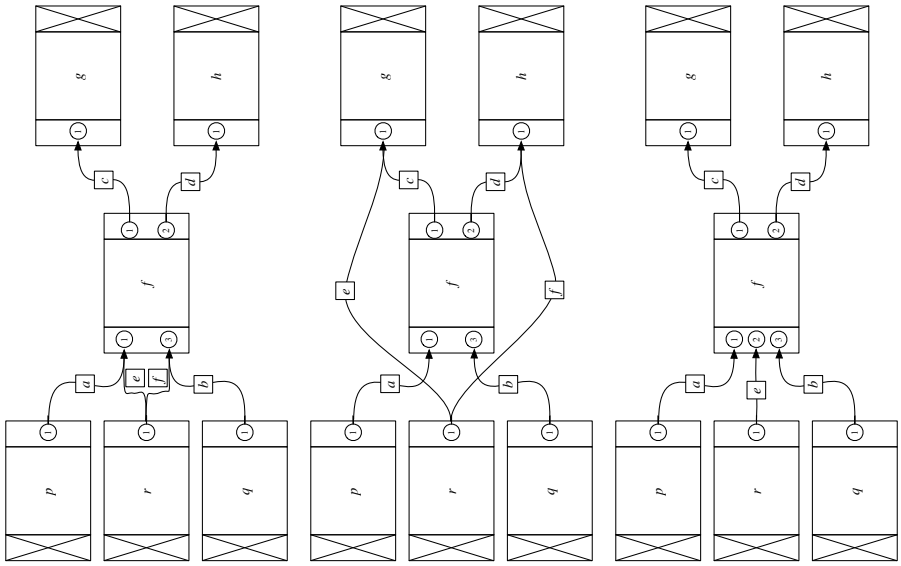


Figure 3.3: Three alternative implementations

1. First, f copies the contents of all $\langle i|f$ into its local memory as the i-th arguments of the function call of f.
   The fact that copying an entire list of values (one after another) may result in a local copy $x_i := \langle i|f$ at time t but $[x|_j := \langle j|f$ at time t' such that the local input vector copy is not "time consistent". This does *not* matter here since our entire system is assumed to consist of unsynchronised, concurrent components anyway and all functions f are required to be robust against resulting information flow delays.

2. Once a new copy of an entire input vector has been loaded into the private memory, the program to compute $f(x_1, \ldots, x_n)$ is started. Memory required during computation is entirely local and invisible to the outside.
   Channels may at any time access the interface variables and overwrite the contents of $\langle i|f$.

3. After computation, the original input that was copied in the first step is copied back to the second, protected, part of the input buffer: $[i|f := x_i$. The result of $f(x_1, \ldots, x_n) = \langle y_1, \ldots y_m \rangle$ is copied componentwise to the protected part and from there to the volatile output buffer: $|j\rangle f := (|j]f := y_j)$
   During this step, any other (reading) access to these memory cells is prohibited as it may contain inconsistent value tuples.

4. Now, $\langle [i|f \rangle_{1 \leq i \leq n}$ and $\langle |j]f \rangle_{1 \leq j \leq m}$ represent a locally consistent and specification-

correct input/output pair: $f([1|f, \ldots, [n|f) = \langle |1]f, \ldots, |m]f\rangle$ whereas $\langle i|f$ may already have been altered by some other process during step 2.
Then, f may start again at step 1.

So if we want to record the I/O behavior of f we take a snapshot of the shadow copies (and while taking such a snapshot, f may not alter them as in step 3). The safest way to do so is to add a special channel (or interrupt) signalling all modules to communicate their shadow copies to the snapshotting process where each module may listen for this interrupt signal during any of the steps 1, 2, and 4.
Still, the shadow port values connected through channels need not be the same due to signal travel time and because some module may still be busy with computing old input values. Therefore, there is no "*global consistency*". A first idea to overcome this problem would be to propagate the interrupt signal through the entire system over the ordinary channels together with their corresponding values. But this does not suffice as one can see in the following, more complex example in figure 3.4: Module f receives such a signal twice; once directly from p (with a corresponding value from p) and also from g after processing the input and signal from p. Therefore, depending on the signal runtime along b, the snapshot would either contain the value from p (via k) or g (via d).

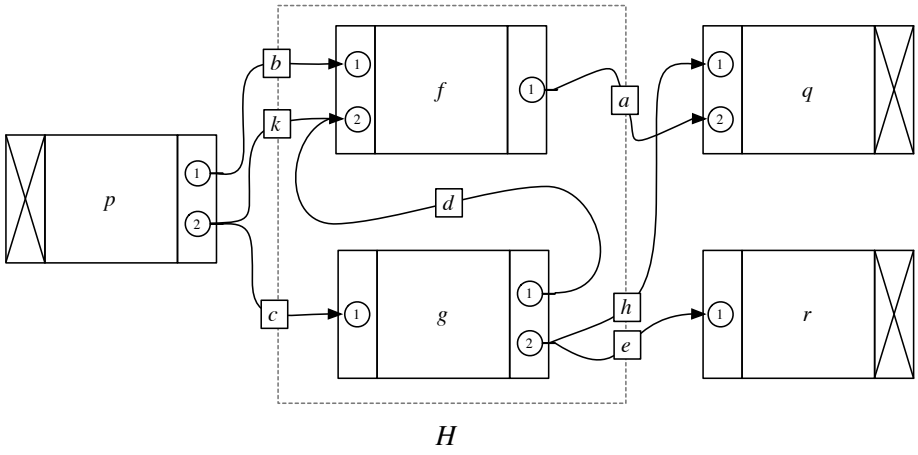


Figure 3.4: A non-trivial module architecture

3.2.4 Aggregates

As shown in figure 3.4, we can compose larger modules or subsystems of modules by analysing their I/O behaviour: p and q, r are intitial and terminal modules and f and g interior modules. The modules f and g appear to work "together" for two reasons: First, they share common inputs from p and second, they are connected to each other (via d).

Therefore they constitute a subsystem (or higher-level module) H with three inputs (via c, k and b) and three outputs (via a, h and e) such that, by some abuse of notation, we may conclude:

$$\begin{aligned} in(f) = \{b, d, k\} \quad &\text{and} \quad out(f) = \{a\} \\ in(g) = \{c\} \quad &\text{and} \quad out(g) = \{d, h, e\}\,. \end{aligned}$$

For H, we have

$$\begin{gathered} in(H) = \bigcup_{m \in H} in(m) = \{b, c, k\} \\ \text{and} \\ out(H) = \bigcup_{m \in H} out(m) = \{a, e, h\} \end{gathered}$$

Channel d appears in both inset and outset—but actually is an *internal* channel (the labelling does not appear on the bounding box around H). This means that (since the outport 1 of g has no target *outside* H), d can b considered an internal channel and the variable holding the value of the outport 1 of g as a *local* variable visible within H only. This analysis, however, is based on the *topology* of the system rather than on an analysis of global data snapshots.

Graph interpretations

ACMEs can (evidently) be represented as graphs with the set S of modules being the nodes and the set of channels the set of edges. To be precise, the set of nodes N is the set of all *ports* with the set of channels C being a subset of $N_{\mathtt{out}} \times N_{\mathtt{in}}$ where the indices indicate whether the corresponding node is an inport or an outport. C can be considered a heterogenous relation $C : N_{\mathtt{out}} \rightharpoonup N_{\mathtt{in}}$ or, alternatively, both sets of nodes together and the resulting graph to be bipartite. The modules themselves establish a family of relations $M_i : N_{\mathtt{in}} \rightharpoonup N_{\mathtt{out}}$, one relation M_i for each module $f_i \in S$. The specification of a module f_i is a function $M_{f_i} : \bigtimes(\operatorname{dom} D) \to \bigtimes(\operatorname{cod} C)$ with $\operatorname{dom} D$ and $\operatorname{cod} C$ being the respective sets of values of elements of domains and codomains of M_i (or, as in section 3.2.1 , *in* and *out*).
In such a graph, we may observe different properties of the system architecture:

- The graph is bipartite, dividing the set of memory cells into those ones holding values of inports and those of outports.
- Channels connect ports, but do not identify port values.
- We assume (though in reality not always satisfied) any pair of nodes to be connected by at most one arrow (i.e. there is no "backup line" where travelling signals may

suggest a time warp due to different delays in different channels): $\langle p, p' \rangle \in C \wedge \langle q, q' \rangle \implies (p \neq q \vee p' \neq q')$.[3]

- A channel $|p\rangle f \mapsto \langle q|g$ for which there is no module such that $q \in \operatorname{dom} M_g = in(M_g)$ or $p \in \operatorname{cod} M_f = out(M_f)$ has "loose ends".
  This notion will become important when discussing interfaces to module system substructures
- Terminal modules do not have outports and initial modules do not have inports.

Identification of substructures

The most interesting property of such a system is to find substructures. We call $H = \langle S', C' \rangle$ an *aggregate* (or subsystem) where $S' \subset S$ and $C' \subseteq C$ is the set of channels that are connected to ports in S':

$$C' = \left\{c : c = |p\rangle f = \langle g|q \text{ and } M_f \in S' \vee M_g \in S'\right\}.$$

If both M_f and M_g are elements of S', then any channel $c \in (in(f) \times out(g)) \cup (out(g) \times in(f))$ is called a *H-internal channel* and an *interface channel* otherwise. If $M_f \notin S'$, then the port q of M_g is an H-inport and p is an H-outport, if $M_g \notin S'$. Ports connected by internal channels are called internal ports; note that internal ports can also be in- or outports. Analysing such systems one will discover three basic types of aggregates:

1. Any set of modules forms an *aggregate* (including the empty set or singletons).
2. A *chain* is an aggregate H with two modules $M, N \in H$ where all H-inports are M-inports and all H-outports are N-outports.
   Aggregates can be transformed into chains by introducing new interface modules bundling all in- and outchannels.
3. A *sequence* is a chain $H = \{M_i : i \in \mathbf{n}\}$, such that there exists a permutation $p : \mathbf{n} \to \mathbf{n}$ such that for all channels $c = |p\rangle f \mapsto \langle q|g \in C'$:

   $$f \notin H \iff g = M_{p(0)}, \quad g \notin H \iff f = M_{p(n-1)}, \quad f = M_{p(i)} \iff g = M_{p(i+1)}$$

   This means that for any module there is a unique "successor" module.
4. A *bottleneck* is a (small) set H' that together with two other (larger) aggregates H and H'' forms a sequence $HH'H''$. It means that all messages from H need to pass through H' before information flow reaches H''.
   Interfaces designed in the course of transforming aggregates into chains are bottlenecks.

[3] However, it is very well possible that f is connected to h directly (fCh) *and* via a third function g. Then, fCg and gCh also imply $f\, C ⨾ C\, h$.

5. *Concurrent tracks* are aggregates that share bottlenecks as interfaces.
   Any two modules of two different concurrent tracks may be connected through a channel.

6. *Parallel tracks* are concurrent chains with identical in- and out-sets.

While these properties can be formalised clearly, they will hardly ever occur in real-world applications but only as weaker versions. For example, two tracks can be "nearly" parallel, if they share only "few" modules.
Furthermore, a system can be decomposed into many different aggregates (in principle, $2^{|S|}$) where every single module or aggregate plays different roles. In figure 3.5, the subsystems have several properties which are summarised in table 3.1.

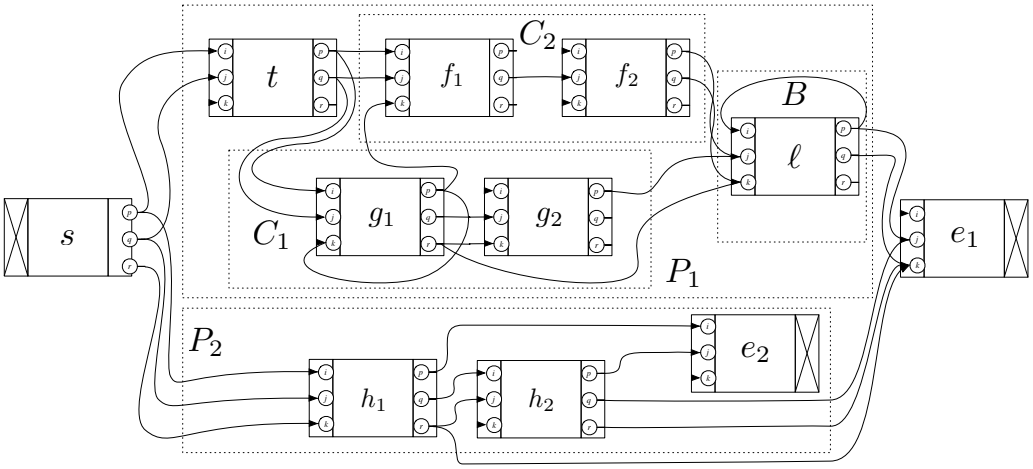


Figure 3.5: Several Subsystems

H	Chain	Sequence	Bottleneck	Concurrent	Parallel	Explanation
B	✓	✓				trivial.
			✓			for C_1 and C_2.
				⊗	⊗	
C_1	⊗					$in(C_1) = in(g_2)$ but $out(G_1) \neq out(C_1) \neq out(g_2)$.
		⊗				$\|p\rangle g_1 \mapsto \langle k\|g_1$.
			⊗			trivial.
				⊗	⊗	$in(C_2) \neq in(C_1)$ because of $\|p\rangle g_1 \mapsto \langle f\|f_1$.
C_2	✓	✓				$in(f_1) = in(C_2)$ and $out(f_2) = out(C_2)$.
			⊗			trivial.
				✓	⊗	with C_1.
P_1	✓					Via t and $B = \{\ell\}$.
		⊗				Since $C_1, C_2 \subseteq P_1$ are concurrent.
			⊗			Since P_1, P_2 are parallel.
				✓		With initial and terminal modules being bottlenecks
					✓	Concurrent without interaction.
P_2	⊗	⊗	⊗			Since, e.g., $\|r\rangle h_1 \mapsto \langle k\|e_1$ and $\|q\rangle h_2 \mapsto \langle j\|e_1$.
				✓	✓	concurrent to P_1 without interaction, and t and e_1 being bottlenecks.

Table 3.1: Properties of Subsystems

Chapter 4

Logic and Algebraic models of concurrent processes

In this chapter we finally present several formal approaches to describing concurrent systems. We point out several rather evident parallels between algebraic and logic descriptions of a program's semantics and the module representation in chapter 3. We also discuss how these methods might be useful to overcome the obvious limitations of the semantics and verification method described in chapter 2. Finally, we will formulate several concrete open questions that require further research towards devising a suitable theory for analysing module structures in a well-defined formal framework.

4.1 Concurrent Kleene Algebra

Similar to our considerations about the different pros and cons of different flavours of program semantics, [35] in their introduction motivate a generalisation of concurrent Kleene Algebra (CKA, see also [31, 33, 34, 36]).
The foundation of CKA is the notion of *events* from a set E (i.e. changes of the environment as a result of a program or program instruction). Here, by environment, one refers to the memory (state) specifying the variables' values that a program operates on. A program that "triggers" events hence changes this environment by assigning (different) values to variables; i.e. by accessing memory. The set of events triggered by a program forms a program *trace* (sequentiality is obtained by time-indexing the elements of a trace). Vice versa, a program (or rather its extensional semantics) is a set of (all) such traces. For the remainder of this rather informal summary of CKA, we will not distinguish between a program P and the set of its traces; but we use p to denote an element of P; that is, a single trace which is an "instance" of P.

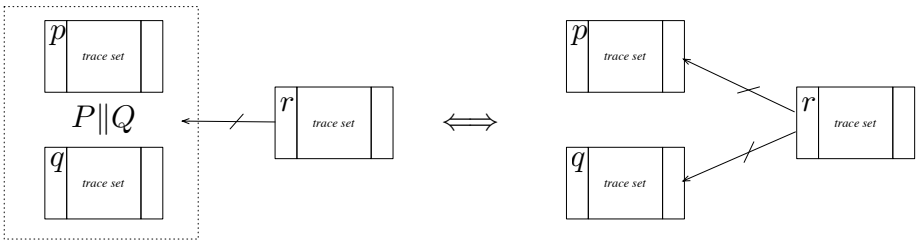


Figure 4.1: Independence conditions in CKA

4.1.1 Composing complex programs from simpler ones

Given (`WHILE-`) programs $P, Q, R, \ldots$, one defines several operations on them (resp. on individual traces p, q, r from their trace sets) to construct more complex ones (cf. modules and systems of modules).

Dependency. Just as in any other environment consisting of several individual processes or agents, the flow of information, data, control or goods *depends* on the overall (logic of) computation or construction. Cars and motorcycles can be built in parallel but both depend on steel, and steelmills can be operated in parallel to both car and motorcycle factories—as long as at any time there is a sufficient supply of steel. In the assembly of a distinct car, the drivetrain and the chassis are built in parallel until they are joined but in building the drivetrain, first the clutch is attached to the gearbox and then the gearbox to the engine, and the painting of the chassis first requires a priming coat, then a coat of lacquer and finally a coat of varnish.
Hence, a process q depends on a process p, if q requires some output of p in order to work. More formally, dependency is an irreflexive relation "$\rightarrow$". Since programs and, hence, traces and process can be composed it is reasonable to take into account the transitive closure $\rightarrow^+$ of dependency: If $p \rightarrow q$ and $q \rightarrow r$, then $p \rightarrow^+ r$ models "indirect" dependency (no matter whether r depends directly on p, too). it seems reasonable to consider the transitive closure $^+$ of $\rightarrow$. In ACMEs, the assumption of irreflexivity is violated because there no restriction on the "channel direction": Loops are allowed—which implies that p may depend on p.
[34] use a bilinear *independence* relation $p \not\leftarrow q$ to state that p does not depend on q and define *aggregates* of P and Q by $p \cup q$. Using the bilinearity property, one then concludes that an aggregation of two programs P and Q is independent of R iff both P and Q are independent of R: $p + r \not\leftarrow r \iff p \not\leftarrow r \wedge q \not\leftarrow r$. The different kinds of aggregation correspond to the according composition operators. An example is shown in figure 4.1.

Sequential composition. $P;Q$ is the program that first executes P and then Q. Note that on trace level, $p \in P \wedge q \in Q \not\Longrightarrow pq \in P;Q$ since there might be quite many

traces ending in a state on which q is unable to continue or successfully terminate. As an example, consider P to be a program that decrements a variable y until its value becomes 0 and Q to be a program that computes the quotient x/y. Hence, $p;q$ is a trace of the sequential composition $P;Q$ if no event in p depends on any event in q, where *dependence* is a transitive relation $\longrightarrow: E \rightharpoonup E$ that indicates data or control flow "from" one event "to" another. Should, in this example, an event in p depend on an event in q, this would require to execute q *before* p which is impossible as the sequential composition requires P to be computed before Q. In other words, at any time, all events are determined by events of the past only. In concurrent programming, programs cannot only be executed one after another but also in parallel and, whenever carried out in parallel, they may or may not interfere.

Concurrent composition $P * Q$ does not require mutual independence of p and q such that $p * q$ is simply the set of all pairwise disjoint unions of traces $p \dot{\cup} q$.

Parallel composition $P \| Q$ requires a different kind of mutual disjointness. P and Q can be executed in parallel if neither one depends on the other; i.e. some p is in $p \| q$, if no event in p depends on any event in q and vice versa.

Alternation is the third kind of mutual exclusiveness: $p?q \in P?Q$, if either $p = \emptyset$ or $q = \emptyset$. It simply means that either P or Q comes to execution but not both of them.

With its origin in Kleene algebra, one focuses on the algebraic formulation of program properties using operations on programs such that the elements of the base set that we talk about are sets of traces.

The algebraic consequence is that one requires, for example, annihilators w.r.t. sequential composition. The program false is defined as the empty set of traces $\{\}$ and the program skip, is the set of exactly one trace which is empty: $\{\{\}\}$. Hence, false does not have any trace which means that it is impossible to execute. skip can be executed since it has a trace, but this trace does not contain even a single event. Therefore, skip can be successfully executed with the result that nothing happens (for the execution of something would always result in at least one event).

Similarly, $P \| Q$ denotes the *parallel composition* where P and Q may be or are executed concurrently without any further restrictions or hidden fallacies since P and Q are assumed to be independent (i.e. there is no data flow from P to Q or vice versa).

Things are a bit different with $P * Q$, the concurrent composition allowing dependencies between P and Q. The semantics of the different operators are defined in terms of corresponding trace composition operations: $P * Q$ is the set of all unions of P- and Q-traces that do not share common events, $P;Q$ is $P * Q$ restricted to those traces where the P-part does not depend on Q, $P \| Q$ is $P;Q$ restricted to those traces where q does not depend on p, and so on (see [33, 34]).

Intuitively, the resulting diagrams already suggest a strong connection to our notion of bottlenecks defining starting- and endpoint of parallel sequences (see figure 3.4). Similar diagrammatic definitions and proofs were also used in [88].
Non-trivial laws of CKA include the obvious fact that $P;Q \Longrightarrow P * Q$, $P;(Q * R) \Longrightarrow (P;Q) * R$ and the exchange law $(P * Q);(R * S) \Longrightarrow (P;R) * (Q;S)$. Here, the implication arrow $\Longrightarrow$ is used for an easier intuitive understanding: Any trace of a program in the premise is also a trace of the program in the conclusion (in [34], "$\Longrightarrow$" is defined as the natural order $\leq$ on the semiring underlying the definition of CKA).
What appears counterintuitive at a first glimpse can be explained quite clearly by ACMEs: If it is possible to concurrently (in terms of ACMEs, i.e. interaction is allowed) run P and Q and then concurrently run R and S, then there is also a way to concurrently run the sequences $P;R$ and $Q;S$. Note that $(P * Q);(R * S)$ means that there is a sequence of changes of an initial environment such that we end up in some final environment. P, Q, R and S satisfy the laws of CKA in the above sense, if for every such possible pair of initial and final environments there is a corresponding sequence of changes of environments such that $(P;R) * (Q;S)$ also "accepts" this pair as initial and final environments.
However, the connections between CKA and ACMEs are not as clear in their entirety as suggested by intuition; for example when taking into account (nondeterministic) message passing (that is, event sequences and, hence, traces) this clear and sharp algebraic definition cannot be transferred to ACMEs one-to-one.
The more recent developments in CKA, [35], eventually lead to an *interface model* by abstracting from traces and using a representation based on *graphlets* and *statelets* which roughly correspond to module structures and connections between them.
Similar to the ACME-paradigm of *interfaces* as links between in- and out-ports, arrows in graphlets are described by their source and target events; similar to the ACME-notion of information flow within the boundaries of an aggregate, we have *internal* arrows; and similar to the ACME definitions of different types of aggregates, we can define *dependencies* and combinability of sets of events (i.e. modules) and, subsequently, give a clear definition and semantics of sequential and concurrent composition.

4.1.2 Concurrency and Separation Logic

[65] describe the close connection between CKA and a logic for reasoning about concurrent processes called *(concurrent) separation logic* (CSL), [9, 75].
One of the most important characteristics of separation logic is that it distinguishes between the valuation of "global" and "local" variables (to speak in terms of ACMEs). Formally, the Hoare-calculus is endowed with an additional rule

$$\frac{\{\!|\, p \,|\!\}\, P \,\{\!|\, q \,|\!\} \quad \{\!|\, r \,|\!\}\, Q \,\{\!|\, s \,|\!\}}{\{\!|\, p * r \,|\!\}\, P \| Q \,\{\!|\, q * s \,|\!\}} \text{ (Concurrency rule)}$$

where $*$ denotes the *separating conjunction*: $p * r$ is satisfied if there are two disjoint memory regions h_p and h_r holding the values of variables occurring in the expressions p and

r such that each of them can be satisfied on one of the corresponding memory regions. So whenever P and Q do not touch any common variables, or if the variables touched by both are irrelevant with respect to the satisfying conditions of p and r, the system is "safe" in the sense that there are no interfering memory access operations while executing P and Q concurrently. This, however, is not the case in reality and in particular, is even intentionally violated by the idea behind ACMEs. [66] write: *[...] in less simple situations, such as when [...] data is referred to indirectly via addresses, or when resources dynamically transfer between program components, correct separation is more difficult to maintain. Such situations are especially common in low-level systems programs whose purpose is to provide flexible, shared access to system resources.* Also, *[t]he essential point is that [a standard workaround] does not cope naturally with systems whose resource ownership or interconnection structure is changing over time.* This will play an important role in section 5.3.1. The beauty of separation logic is grounded in the fact that all requirements on program behaviours are *hidden*; i.e. the above rule *presupposes* P and Q to be implemented in a way that avoides interference. If we imagine a bank to be a module that maintains storage (accounts), a single customer (client) is not allowed to access an account directly (by tampering with the data describing his assets) but needs to ask the bank for a money transfer action (i.e. rewriting the contents of two accounts). This way, it is only the bank that has access to the variables although any client may ask the bank for performing the action he would like to execute himself if he had the right to do so. This requirement contradicts the fundamental idea behind ACMEs: an ansynchronus, concurrent and mutually independent access to shared data without any warranties or obligations. One simple example demonstrating the problems of using CSL to describe ACMEs is the

$$\frac{\{\!|\, p \,|\!\}\ P\ \{\!|\, q \,|\!\}}{\{\!|\, p * r \,|\!\}\ P\ \{\!|\, q * r \,|\!\}}\ \text{(Frame Rule)}$$

stating that if P satisfies p and q to be valid pre- and postconditions, P also satisfies p and q when monotonically increased by some r. With P having side effects (like results not observed by q, e.g. P_2 affecting $|i\rangle\, e_2$ in figure 3.5) or a non-deterministic behaviour due to weak synchronisation (like g_1 in C_1 writing to $\langle k|\, f_1$ which eventually leads to conflicting access to $\langle j|\, l$ and $\langle k|\, l$ by C_1 and C_2), the frame rule obviously does not hold. Similar considerations apply to the mutation statement ":=" where for two concurrent ("‖") mutations on the same memory cell there does not exist a sufficient precondition at all.

Yet, the requirements on P and Q can be expressed in terms of *hypotheses* (corresponding to the presupposed behaviour). Adding descriptions of program and environment behaviour and coping with them within the calculus syntactically, results in a huge increase of notational complexity. This contradicts the idea behind hiding side-effects and the clean formalism and also hampers manual or automated inferencing. Preliminary work on explicit encoding of hypotheses or extended annotations by Dang and Möller has been suspended due to the questionable added value of overly syntactical complexity.

4.1.3 The Temporal Logic of Actions (TLA) and distributed snapshots

Lamport's Temporal Logic of Actions (TLA), [43], offers a method to cope with such effects through the notion of *valid actions* that applies to actions (or predicates specifying modules) and pre- and post-states whatever the variable instantiations are.
In other words, it gives us a logical description of the semantics of an action—which is exactly what we are in search for when trying to understand ACMEs without a proper specification.
The problem here is to understand *which* (parts of a) state are to be monitored and *when*. Even though different in details of formalisation, [44] give a definition for (global) states of distributed systems that allows treating the problem of validating *stable properties*. The underlying model of distributed systems and channel (wiring) is much simpler than ACMEs, but takes into account queuing of messages in channels. In addition to this, events occurring in a component may affect at most *one* of the outward channels which (trivially) connects to at most one other component. As a result, messages cannot be lost for two reasons: They are stored within a channel until read by the receiver (who then may decide whether to work with it or discard it), and it is easy to implement a token-based synchronisation method that guarantees the inputs of a component to form a valid and well-synchronised input pattern.
The idea of taking snaphots in ACMEs was based on locally keeping copies of valid input-output pairs in each component (satisfying "local consistency") as elements of global snapshots (triggered by an interrupt). Still, this did not guarantee global consistency (for the different components might have different histories of processed input data). In Lamport's approach, global consistency can at least be determined by verifying that all module and channel states "agree" on a common state of communication flow: State descriptions of a component and an outgoing channel can only live together in a consistent global state, if, roughly speaking, the number of messages that left the component equals the number of messages that entered the channel. The reverse case holds for incoming channels; and together, both requirements form a snapshot policy of a "snapshot" token traveling through the network to ensure a snapshot without "time warps" in it, which is implemented as a *marker*-based global state detection algorithm. Examining permutations of component execution sequences allows to determine which resulting consistent states then satisfy a stable property; in terms of ACMEs, a possible system specification is a property that is satisfied by (most) snapshots (and by all globally consistent snapshots should there be any).
What has been observed at a rather low system level has, among other reasons, finally led to the introduction of so-called *stuttering steps* in [43]'s temporal logic of actions. Its modal logic description, together with the formulation of liveness and fairness, eventually leads back to our ideas of formalising ACMEs in concurrent Kleene algebra and thus completes the picture of the proposed project.

Chapter 5

Current and future work

The general picture which we consider to motivate future work is that of a complex system of proccesses evolving over time. Figure 5.1 shows an example of system changes over time and the changes of available specifications (lower case letters) of involved processes (uppercase letters). The task ahead is to explain the final system failure where several specifications were lost and components of the system have been replaced without documentation or intermediate verification.

5.1 Evolution until failure

Figure 5.1 illustrates several common phenomena that occur in long-term application and evolution of complex systems.
Initially, the system was entirely specified (a, b, c and s) and correctly implemented (A, B, C and S). The system behaviour was as intended, as expected and as specified.
The first change during S's lifetime was when glitches occurring in B (for example by wear) were locally fixed. The resulting D (seemingly) worked within b's allowance. At the same time, the specification a for A was lost.
Second, C has been replaced by E. A common reason is that, for example, C is a component supplied from an external company (like libraries in the case of software engineering or machines for which consumable supplies are not deliverable any more) and requires a replacement. The supplier delivers a guarantee that E's behavior is equivalent to that of C but does not deliver a specification (e.g. due to licenses) or source code. It is unclear, whether the equivalence of E to C is local or with respect to s. However, due to the promise of E's correctness, the specification c is discarded.
A third stage is reached, when due to occasional global errors, E gets replaced by F but a backup copy of E is retained
Finally, S delivers unpredictable output and ends up in an entire system failure. An even worse case includes the loss of s: Since S "more or less" behaved in the "usual" way, it

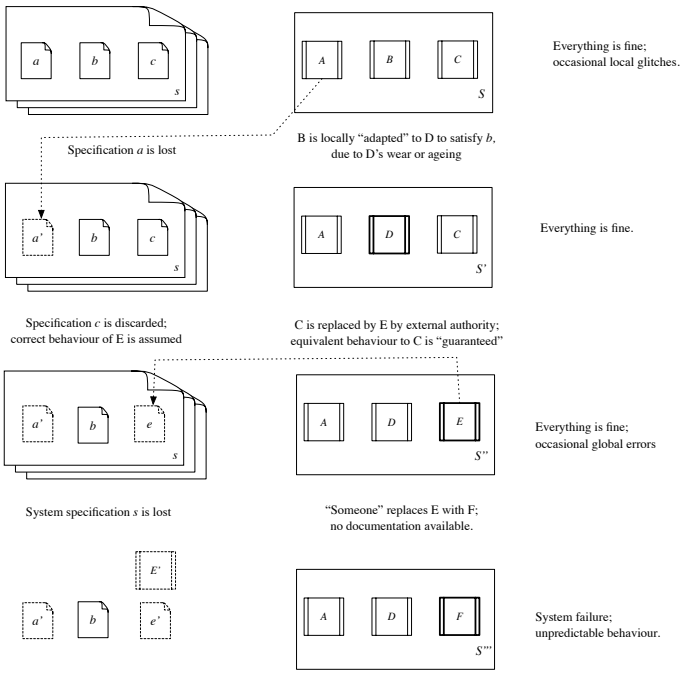


Figure 5.1: Evolution of a complex system of processes

seemed unnecessary to keep s because routine and the long-term observed correct I/O-behaviour of S is taken as a sufficiently precise specification.

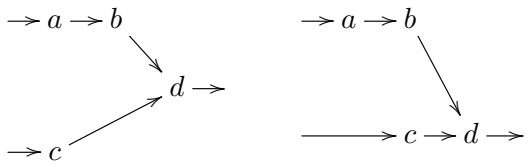
5.2 Modelling Concurrent Systems

5.2.1 Theoretical Foundations

Chapters 3 and 4 described and exemplified the similarities between a practical and real-world driven view on concurrent processes (ACMEs) and recent theoretical approaches towards the algebraic specification and verification of concurrent processes (CKA).
However, it remains to rigorously formalize and represent ACMEs within CKAs. Having done so, the next step is to formulate all rather informal module and system properties (such as "bottlenecks", "interfaces", "aggregates") in CKA. Finally one would like to implement several example ACMEs in CKA:

1. A first experiment is to formulate a simple and strictly sequential architecture such as FBDs, (see section 5.2.2).

2. Based on first experiences, the next step is to model a simple concurrent example like the phototaxis example from section 3.2.2 and figures 3.1, 3.3,



Both diagrams are translated into the program $a; b; c; d$.

Figure 5.2: Programming PLCs

3. Eventually, more complicated structures, where the one in figure 3.4 can be regarded a rather advanced example already, shall be considered.

Recent work of Hoare et al. on unified models and laws for concurrency and distributed processes is directed at a more abstract description of concurrent systems. It aims at a graph-based understanding of dependencies between processes and generalizes previous work on graphlet-based representations of program traces, [33, 34, 32], offering a closer linkage to CKA.

5.2.2 Applications

Phototactical agents and groups thereof can be simulated, [42], and have been successfully implemented using real small robots [37, 26, 38, 83]. Every single robot can be considered an isolated process of a concurrent system.
Also, in [18], the authors describe an approach to verification of programmable logic controllers (PLCs) using an interactive theorem prover (KIV, see [73]). PLCs can be programmed using a variety of different languages; one of the most prominent is a graphical programming language (FBD, Functional Block Diagrams, see [5]) similar to our diagrams for specifying ACMEs. The most prominent difference is that logic gate data flow is strictly serialized during compilation into machine code. Also, diagrammatic specifications may not contain loops (due to serialization). Serialization is achieved by translating the data flow chart into a clocked sequence of a gate-by-gate evaluation. Compilation always ensures independence (cf. section 4.1.1) by a simple trick: The "development tool" is a graphical interface that only allows connecting gates from left-to-right and top-down (see figure 5.2). Hence, loops cannot be designed by syntactical restrictions. Yet each such program defines a single "cycle" as it is repeated over and over with register variable contents preserved from one execution cycle to the next.
The authors already point out an extension of their FBD-restricted approach to other programming languages as well as an upscaling of the programs considered. Model checking for verification of modern PLCs is clearly infeasible due to the state space explosion (no matter whether one deals with single-chip-PLCs or more powerful systems). With increas-

ing complexity of the program one might like to consider state logic for programming and a model in modal (temporal) algebra for verification.
Concluding, simple autonomous agents and cooperating robots controlled by PLCs in a concurrent setting appear to be an interesting real-world application for ACMEs at the microscopic (embedded, "on-chip" concurrent processes) and macroscopic level (cooperating agents and robots).

5.3 Learning about concurrent systems

In most cases, model checking, see [47, 2], is out of the question as a method for verification of complex or even concurrent systems. But being a method based on "systematic search for faulty behaviour" it is well accepted in areas where a system is considered to be correct as long as no error is observed. Then, testing or partial model checking is considered a proof of correctness unless an error occurs. For *small finite* problems, verification in the strong sense can be achieved by model checking: A system is correct, if *every* model (possible system state) is consistent with the system's specification.
Model checking has been applied for verification of PLC programming using a wide variety of model checkers such as Uppaal, [84], NuSMV, [68], [mc]Square (now Arcade), [80] and SPIN, [89], using proprietary specification languages (Uppaal, PROMELA) or the PLC programming language IL directly. Most results report that verification by exhaustive state space exploration is *possible*, some even state that feasibility is *scalable*. Still, as [89] write in their conclusion, state space size "can" explode. Considering the *simplest* PLC (for example one of the Simatic series), its capabilities *exceed* the requirements for the example applications in the publications mentioned above by far so it is quite likely that state space *will* soon expand beyond feasability. Clearly, the same consideration applies to any other system of autonomous agents or robots.
Apart from the approach to developing verified software from scratch (see section 5.2), the real world task is to discover components or configurations in which errors occur. According to sections 1.3 and the general layout of ACMEs as described in section 3.2.4, we want to infer logic specifications of running systems that produce unwanted failures.
By observing and collecting data about an agent's or robot's behavioural data one establishes a *sample* from which machine learning techniques try to infer more general descriptions. In our context it is reasonable to consider *logic* based approaches, [58, 71, 51, 52, 55], rather than statistical, probabilistic or evolutionary ones, for what we are aiming at is a logic or relational abstract description of the observed data. The data used to build a sample for successful hypothesis induction has to meet several requirements that depend on the chosen method. Nearly all methods are vulnerable to *noise*.
At a very abstract level, noisy data is the result of an observation that does not comply with the event. The usual understanding of the term "noise" is based on

- the presence of outliers (given that the method for identification of outliers is com-

patible with the model),

- erroneous labelling of observations due to external errors (such as wrong sensor readings),
- the presence of data supporting a wrong assumption about system.

The latter phenomenon is actually needed for two reasons: The first reason is that overly precise data may lead to overly precise, so-called *overfit*, hypotheses unless one spends effort in artificially weakening the observed data (which again creates bias). The second reason is that we *want* wrong hypotheses. If an analysis of induced systems descriptions reveals characteristics that contradict the intended system behaviour we have found exactly what we were looking for: A candidate for causing the system failure.
For this, one requires a delicately well-balanced amount of "suitable" noise in the sample. This is an unsolvable problem, because we would need to know the right incorrect hypothesis in order to guarantee the suitability of the sample in advance.
As a consequence, we want to create data samples statisfying different consistency criterions. Hence, we shall explore how to combine Lamport's theory of distributed snapshots [44] and its logic [43] with separation logic and, especially recent developments in concurrent models of concurrent program models and its treatment in Kleene Algebra, [35, 67].

Research tasks

We conclude and summarize the most interesting tasks ahead:

1. Complete existing approaches for a uniform and compatible relational specification of ACMEs.
2. Find an embedding functional description of ACMEs in CKA.
3. Examine approaches of Separation Logic and Temporal Logic of Actions .
4. Augmenting Separation Logic with additional annotations describing pre- and post-conditions of ACME-modules.

5.3.1 Learning about concurrency

The term "concurrent learning" has two different interpretations both of which give rise to open research questions:

1. How can one express and learn about concurrent processes?
2. How can one *concurrently* learn about such processes?

The first one adds to learning what concurrency adds to sequential processes: Sequence learning has been in the focus of machine learning research from the late 1990s on. The word "sequence" has two meanings in machine learning: The first one refers to a *temporal* arrangement of single observations whereas the second one refers to a *spatial* arrangement of atoms that together form a single observation.
Most methods employed for temporal sequence learning HMMs, probabilistic approaches, recurrent artificial neural networks or windowing techniques, [86], in order to find finite (small) models for describing and predicting (infinite) sequences of observations.
In addition to HMMs, Bayesian methods and artificial neural networks have been used for the discovery of structural similarities of subsequences (or "strings"). Also, support vector machines have shown great potential, e.g. [85, 3] (in contrast to temporal sequence learning). Massively parallel approaches, mostly inspired by evolutionary computing, [19], have been employed to find (regular) patterns and induce suitable pattern matching methods to solve the so-called alignment problems, [63, 46].

In this section, we shall treat the question how one can discover knowledge about concurrent systems. Concurrent approaches to knowledge discovery will be dealt with in section 5.4.
In the following, a variable X is used to denote a single module or process; $\overleftarrow{X}$ denotes the arguments to the function f_X computed by X and $\vec{X}$ its output. Lowercase letters x refer to X's specification and $\langle R|\,\overleftarrow{X}$ and $|R\rangle\,\vec{X}$ describe modules that directly deliver or receive values from is$X$ as specified by a dependency relation R. For the sake of readability we do not distinguish between a variable and its value as long as clear from context. *Sampling* means to collect a series of snapshots over time. If we consider S to be the set of all modules X_i with $i \in \mathbf{n}$, then a complete snapshot would be a recording of all current values of

$$\left\langle \overleftarrow{X}_0, \vec{X}_0, \overleftarrow{X}_1, \vec{X}_1, \ldots, \overleftarrow{X}_{n-1}, \vec{X}_{n-1} \right\rangle.$$

Depending on the number n of modules and their respective arities, a single snapshot may already consist of a significant amount of data—which also requires a significant amount of time to be recorded (during which at least parts of the system if not the entire system is forced to pause current processes). For two modules U and V, the I/O behaviour of $U;V$ is described by the tuple $\left\langle \overleftarrow{U}, \vec{V} \right\rangle$. However, this cannot be observed in a setting described above: If the snapshot is consistent with respect to the modules, then $f_U(\overleftarrow{U}) = \vec{U}$ and $f_V(\overleftarrow{V}) = \vec{V}$, and if the snapshot is consistent with respect to communication presupposing instantaneous signal travelling, $\vec{U} = \overleftarrow{V}$ but not necessarily $f_U(\overleftarrow{U}) = \vec{U}$.
Assuming that U delivers at least some input to V ("$\vec{U} \cap \overleftarrow{V} \neq \emptyset$") the question is to which extent V's output relates to or is based upon U's output. The result of analysing a sample over $\vec{U} \times \vec{V}$ can be taken as a hypothesis for the semantics of an operator "$U;{-}V$" or its dual $-$; describing extremal conditions on the value ranges.
Finally, if we assume U delivers to V and V to W, a sample on $\vec{U} \times \overleftarrow{W}$ delivers an "external view" on V. Since most modules will take and deliver to several models this simple view

would be to narrow and one would have to consider snapshots of all modules that are connected to V. This requires knowledge about the dependency relation (by specification or by system analysis). On the other hand, it also offers the possibility of identifying redundancy in the system: If $|R\rangle\, \vec{U} \cap \overleftarrow{V} \subseteq (\bigcup_{i \in \mathbf{m}} |R\rangle\, X_i) \cap \overleftarrow{V}$ the information supplied by U to V is already included in the information that V takes from X_i. Hence, U is X_i-*dispensable* with respect to V (for the notion of relative dispensability by redundancy in information systems, see [59]).

Having collected data describing the actual behaviour of a (faulty) system, the next task is to infer hypothetical specifications of the system and its components. The approximation of a function implemented by a certain process corresponds to the traditional machine learning setting:

5.3.2 Specification recovery

In our example, no-one ever observed that A has been changed which is why we would assume that A actually is the same program or process that was originally developed to implement a. The specification a was lost very early such that one would like to *learn* a specification a' of A. Since we need a' to reason about A and its behaviour within S, we need to use a logic based knowledge discovery method. The data required to learn a' is a sample of A's behavior:

$$\mathbf{s}_A \subseteq \left\{ \left\langle \overleftarrow{A}, \vec{A} \right\rangle : \overleftarrow{A} \in \operatorname{dom} f_A, \vec{A} \in \operatorname{cod} f_A, f_A(\overleftarrow{A}) = \vec{A} \right\}.$$

The quality of $\mathbf{s}_A$ depends on the sampling method: Should an example be recorded by a memory snapshot, where some memory cells used for storing $\overleftarrow{A}$ have been accessed by another process while the memory for $\vec{A}$ remained untouched, it could be that the values do not correctly represent f_A's behaviour. Hence, the method of snapshotting determines the kind and amount of noise present in the sample.

5.3.3 Approximate verification by comparing abstractions

The replacement of B with D took place "locally" and "gradually". D is not explicitly documented and it is even unknown to S that B has been replaced by D. It is assumed that D behaves "correctly": Not any component has ever observed a situation in which D's output caused an error (including D itself). Instead of proving that D satisfies b, we apply the same method as for learning a specification for A: From $\mathbf{s}_D$, we induce d. Should $d \leq b$ (where $\leq$ means refinement), we may infer D is most likely correct with respect to b. However, if the sample is too precise, the hypothesis d may be overfit, too. Therefore it seems reasonable to force further generalisation of d by adding examples to $\mathbf{s}_D$ that have *not* been observed but which still comply with b.
Similarly, if $b < d$, we cannot safely deduce that D does *not* comply with b, because d could be an overgeneralisation caused by a noisy sample.

On the other hand, being a propor specification, b should clearly describe the weakest precondition for B to perform correctly and it should also describe a strongest postcondition for the results that B delivers. Therefore it has to be evaluated, whether they can be used to define upper and lower bounds for enlarging a sample in order to avoid overfitting and de-noisifying a sample in order to avoid erroneous overgeneralisation. Then,

$$
\begin{array}{lll}
\mathbf{s}_{D(b)} \subseteq & \mathbf{s}_D & \text{|: original sample} \\
\cup & (\langle b| \times \operatorname{cod} f_D) & \text{|: all "situations" satisfying the weakest precondition of } b \\
\cup & (\operatorname{dom} f_d \times |b]\,) & \text{|: all "results" satisfying the strongest post condition of } b \\
- & (\overline{\langle b|} \times \operatorname{cod} f_D) & \text{|: all invalid situations} \\
- & (\operatorname{dom} f_d \times \overline{|b]}\,). & \text{|: all invalid results}
\end{array}
$$

This training sample generation is a "verified" version of what otherwise is known as co-training, [4]. With too little training data, one learns several classifiers on even less data. Obviously, the results are *weak* hypotheses and, due to their disjoint and very small training sets, *unstable*. This justifies an application of both boosting and bagging techniques, [79, 8]. Actually, co-training combines them by selecting a "best" predictor learned from the small sample sets and then use this to generate more, artificial data on which the entire process is iterated.
[57] outlines an approach to suitable sample decomposition using a relational analysis.

5.3.4 Local learning by global feedback

The replacement of C with E combines the former two phenomena: First, c has been discarded as well, so that a verificiation of E against c as in section 5.3.3 is impossible. Also, C has been deleted, so that there is no chance for a comparative run-time analysis of C and E. However, it *is* possible to learn a specification hypothesis e as described in section 5.3.2. But as e is induced from E's behaviour, it cannot be used to verify E.
A further effect is that S produces occasional errors. We assume that these errors are due to the latest system change and they are independent from A, B and D. This is a strong assumption for several reasons. For example,

1. in the context of ageing systems errors may be caused by any of the involved components at any time, so that the co-occurrence of an error with the introduction of E is pure coincidence and the error is caused by any of the other components.

2. when observing an error of a system S'', it may well be that the cause occurred in S or S' already; i.e. the error does not all relate to S''.
   Suppose that A in S produces output that is processed by B and, in S', by D. Assume further, that D reads A's output from a queue much more slowly than new data is added to the queue by A. Now, in S'', C is replaced by E but it is data that A sent in S or S' already to B or D and that only now causes an error in in S''.

Though very strong, the assumption is justified as it reflects the real-world: Recent changes that coincide with errors are usually assumed to be the cause. Furthermore, in our example, we have no evidence that A has been tampered with and we are guaranteed that D is correct w.r.t. b, hence there appears to be no other explanation for an error of S'' other than by E.
As already mentioned above, the method described in section 5.3.2 may deliver e'. This hypothesis can *only* be compared to b or s if E is somehow connected to D or S''.
Hence, this situation is very hard to cope with as there is nothing left that we could use as a reference to validate E. A bail-out procedure might be constructed as follows: We observe all those snapshots covering all modules and variables that E is connected to; either directly or through (several) other modules(s); including $\overleftarrow{S}$ and $\vec{S}$. Any snapshot that does not satisfy s is an element of an *error-sample*. On this error-sample, we test (here: A) or, if possible, verify (here: D) the correctness of all involved modules ecxept E. If all other modules perform correctly or if the recorded data is covered by the specification, we assume this snapshot to carry information about an error caused by E. A weaker constraint is that a snapshot captures a possible failure of E if it cannot be shown that other modules do not perform correctly. The weakest support for an error of E is given by a snapshot that also carries evidence for failure of other modules. Then, we can "*co*"-learn a description when E creates errors. Should this error-hypothesis cover (nearly) *all* elements of the unrestricted error sample, we know that whenever S'' produces an error, E produces an error, too. As the reverse implication cannot be inferred, we cannot reliably identify E as the error source but assuming D is correct with respect to b and A never has changed, we have at least strong evidence.
Being just a "bail-out" solution it seems to be overly complicated and it is questionable whether the amount of effort invested pays off. Similar to our sample descriptions in the previous sections, we want to give a brief impression of the sample construction process described above. For this purpose, assume that $\langle S''| \overleftarrow{E}$ denotes all modules whose outgoing connections are incoming connections to E; let $|S''\rangle \vec{E}$ denote the set of subsequent modules that directly receive input from E. Then, the memory cells from which we have to sample is

$$\mathbf{M} = \langle S''| \overleftarrow{E} \cup |S''\rangle \vec{E}.$$

We then collect only those instances, for which $\left\langle \overleftarrow{S}'', \vec{S}'' \right\rangle \notin s$; i.e. where we observe a system error in an error sample $\mathbf{s}_{err(S'')}$. From this sample, we *delete* all examples that describe an error of any module other than E and obtain a sample $\mathbf{s}_{err(E)}$. Note that if we end up with too few instances, we already have strong evidence for E not being the sole source of error. Should we have only few but not too few instances the sample can be enlarged by weaking the constraints of deletion (see above). We then induce a "*co*"-specification $\tilde{e}$ from $\mathbf{s}_{err(E)}$ and compare it to s, should $\tilde{e}$ explain some instantiations of s we have found evidence for E causing S'' to fail.

5.3.5 Explaining a system's behavior

Finally, we discuss the case when the specification s is lost. In addition, further modules may have been replaced (here: E by F). The entire system S''' fails: it is still running, but it produces unpredictable results. The possibilities of collecting snapshots of S''' depends on several circumstances:

- Should the modules still be accessible, it is possible to collect local samples. Consistency of the samples that can be used to induce module specifications depends on the snapshotting method applied to the respective modules.
- Global samples of (parts of) the entire system may reveal common memory access and, hence, dependencies between modules (c.f. [35] and ongoing work).

The first task corresponds to the standard learning setting already mentioned above. The second one offers many more opportunities for a deeper system analysis.

Channels. Equal values in modules $\vec{X}$ and $\vec{Y}$ indicate that Y depends on X. Similarly, but according to the weakening of equality to equivalence, a dependency can be assumed if there exists a bijection between partitions of arguments of $\vec{X}$ and $\vec{Y}$. Both require snapshots that are consistent on at least a subset of all modules from which one then chooses pairs X and Y of modules. Under the assumption of consistency it is unlikely to discover transitive extensions of dependencies, for this would require a value and the result of applying a function on this value to be visible at the same time.

Information flow in space and time. With a low sampling rate even the assumption of consistency does not ensure that temporally dependent events cannot be observed simultaneously. On the other hand, should two subsequent events appear in the same snapshot it means that their observation requires *spatial* dependency. As a consequence, learning from such samples might result in a two-dimensional dependency relation as proposed in [30] but here, temporal dependency may occur as spatial dependency and vice versa: should the *same* event be observed several times in one snapshot, it means that this event travelled too fast for the "exposure time" of the snapshotting procedure such that it was recorded twice.

Discovery of dependency and interfaces. Knowledge about the dependency between modules is a neccessity for understanding a system's behaviour. Should we know about the information flow (i.e. spatial data travel or temporal data persistence) across processes (where across means "between" for spatial dependency and "over" for sequences of processes), we are able to induce hypotheses of the involved modules on the information processing (i.e. their semantics). If we don't, the problem is to identify "infemes" (which would be the information theoretic analogon to phonemes or graphemes). They are smallest and indivisible units of system's entire information process. With no knowledge of a system, the system itself comprises its only infeme as it is a "black-box" that receives input from some initial modules and produces output to terminal ones. Sampling the inner states may help identifying smaller black boxes (of a slightly lighter shade of black) inside.

That is, the system is broken into several subsystems each of which again is considered indivisible at this degree of analysis. The subsystems themselves are connected to what they consider their "outside" through their respective initial and terminal modules. These modules (or channels as well) are, of course, internal to the big system. More importantly, they also play the role of such connecting modules of other subsystems and can be called *interfaces*. Hence, by *learning* from the behaviour and internal states of a system one can induce a *refinement* of the systems' specification.

Concurrency. Concurrent memory access and resulting inconsistencies may occur whenever two modules deliver to one and the same module. Repeated occurrence of values $y \in \vec{U} \cap \vec{C}$ is only a weak evidence; should the same value appear in some $\overleftarrow{W}$ whenever it occurs in $\vec{U}$ and $\vec{V}$ we have stronger support support of the hypothesis that U and V are both writing to W. Again, this observation requires an instantaneous and consistent snapshot. On the other hand, concurrency expresses the nondeterministic (but not necessarily disjoint) choice. Hence, a collection of snapshots over time suggesting that $y \in \overleftarrow{W} \longrightarrow y \in \vec{U} \cup y \in V$ is a more reliable indicator for concurrency. In combination, the more often we observe for as many as possible arguments $(w)_i$ in $\overleftarrow{W}$ the fact that its values $(y)_i$ coincide with values of arguments $(u)_j$ from $\vec{U}$ and $(v)_k$ from $\vec{V}$ the more likely it is that U and V concurrently deliver values to W.

5.4 Learning concurrently

Machine learning for knowledge discovery emerged from the induction of logic programs and was a "late hot spot" of AI research in the 1980-90's. Approaches based on the paradigm of neural networks, [78], date back to the 1950's, [50, 76] and [41], but they always suffered from their inability to explain induced hypotheses.

With growing data and increasing amounts of *unstructured* data there emerged the need for (quick) data analysis which resulted in the paradigm of "data mining". As the amount of data increased faster than available computing power, simple but quick statistical and probabilistic approaches became more popular. In the late 1990's for a short period, data *visualisation* even appeared to be the more useful method for data analysis. Knowledge discovery still remained in areas that put a strong focus on the scrutability of induced hypotheses such as in drug or protein analysis or user modeling.

Now, with petabytes of heterogenuous data constantly and automatically harvested, the question is not "do we have data about x" any more but rather "what data do we have" and "what can we extract from all this data". Recently, [20] have put the 90's approach to knowledge discovery by inductive logic programming into the modern context. They also mention the ideas that we have already explored in this report so far; for example SAT/SMT solvers may be used for a greater variety of specification induction using the machine learning setting with labelled examples as input rather than complete formal specifications.

The challenges they note are

1. Can partial specifications be learned more efficiently and then composed into a specification of the entire system?
2. Can large sets of examples be decomposed in order to induce hypotheses in parallel on each set?
3. How can one handle noisy data?
4. Can one possibly learn hypotheses that are easier to understand than those formulated as (logic or functional) programs?

They conclude with a review of Lau's proposition, [45], that has been well-known ever since in user modelling, e.g. [56]: Data to learn from is hard to come by when relying on explicit (human) user interaction.
All of these challenges correspond to the aims of our project proposal:

1. Partial specifications shall be induced for parts of the entire system,
2. sample data should be partitioned to enable a bagging-like exploration of system behaviour,
3. noise is an issue that we shall deal with in the context of snapshotting,
4. scrutability of hypotheses is implied by our relational formalisation.

Lau's proposition does not apply in our case because the availability of data is guaranteed by collectiong snapshots of running systems.

5.4.1 Learning logical and relational sentences about state descriptions

The noun "sentence" in this section's heading has been chosen for two reasons: First, we want the propositions that together form a specification to be (nearly) as easy to understand as natural language. Second, system specifications require well defined formal (domain-) languages to which one assigns a suitable semantics hence yielding a logic. One such language that we shall deal with is the language of relation algebra; the other one is (a subset of) first order logic. And, at the same time, Horn logic is the subset of FOL that is used as programming language in logic programming, hence the name (e.g. Prolog). Also, Horn logic plays an important role in axiomatizability of (mathematical) theories (such as Relation and Kleene algebras).
We now give a brief impression on how to interpret logic learning as search for theory axiomatization. The next two paragraphs deal with the induction of classification rules from factual observations (as applicable to learning specifications of single modules). The third paragraph of this section sheds some light on how to learn *across* and *about* factual knowledge. Again, the formal presentation is kept very shallow. For the algebraic and logic foundations, we refer to [16] as an introduction and [69, 21], whereas details concerning

machine learning and inductive logic programming in particular can be found in [71] and [58].

Specification by propositions over snapshots. As already pointed out in the sectin about ACMEs, modules are implementations of computable functions mapping input values $(x)_i$ to output values $(y)_j$ with $i \in \mathbf{m}$ and $j \in \mathbf{n}$. With f being such a function, a module M is correct w.r.t. to f iff

$$\vec{x}.M = \vec{y} \Longrightarrow f((x)_i) = (y)_j$$

for $\vec{x} = g((x_i))$ and $\vec{y} = h((y_j))$ with computable functions g, h mapping the arguments of f on memory contents. Assuming g, h to be defined such that $\vec{x}$ and $\vec{y}$ do not overlap (see section on separation logic), and g and h can be composed into one function $val : V \to (T \to U)$ that for any variable in $(x)_i \cup (y)_j \subseteq V$ represents its content $y \in V$ at the current time $t \in T$. Hence, assuming consistency of memory snaphots, the correctness condition from above yields for a given point t in time

$$\left(\bigwedge_{i \in \mathbf{m}} x_i(t) = v_i \right) \longrightarrow \vec{y}(t) = f(\vec{v})$$

which obviously is a Horn clause. Therefore (note that v_i in the premise corresponds to $\vec{v}$ in the conclusion), f is a model of all memory snapshots if M is correct w.r.t. f. The connection to a relation algebraic formulation is obvious.

Snapshots and logic and relational sentences for specification. With a given M but unknown specification f, the task is to find some set Φ of Horn clauses for which

$$\forall t \in T : \langle \vec{x}, \vec{y} \rangle_t \in \mathrm{Cn}(\Phi).$$

In addition, Φ should be

- as general as possible, i.e. $\mathrm{T}h(\Phi)$ should be as big as possible,
- consistent.

Expressing generality in terms of theory size is not really helpful as constructions of recursive terms always allow infinitely many instantiations. Consistency is a bit tricky to deal with: Φ is inconsistent, if both $f((x)_i) = (y)_j$ and $f((x)_i) \neq (y)_j$ are deducible for some instantiations of $(x)_i$ and $(y)_i$. But since f is *unknown*, this fact cannot be determined unless the corresponding tuple is an element of the set of collected snapshots. In machine learning one usually assumes a target classifier to be a total function; in the simplest case, a characteristic function. This means that starting off with a sample, one may *create* a sample of obviously *wrong* classifications called a sample of negative examples.[1]

[1] In simple classigfication tasks, $\mathrm{cod}\, f = \mathbf{2}$ is a characteristic function such that $y \in \{\mathbf{0}, \mathbf{1}\}$. In such cases, one is usually equipped with examples for *both* target values of f.

We write a "complemented" pair $\overline{\langle \vec{x}, \vec{y} \rangle}$ to indicate that $f((x)_i) \neq (y)_j$.[2] Then, the notion of consistency may be weakened to

$$\Phi \text{ is consistent } :\Longleftrightarrow \mathrm{Cn}\Phi \cap \{\langle \vec{x}, \vec{y} \rangle_t : t \in T\} \cap \left\{ \overline{\langle \vec{x}, \vec{y} \rangle_t} : t \in T \right\} = \emptyset. \tag{5.1}$$

meaning that the hypothesis Φ must not enable one to infer positive and negative evidence. Finally, we explicitly want Φ to be *incomplete* for a similar reason: As we are dealing with large margins of vagueness there are (infinitely) many propositions for which do not at all have sufficient knowledge justifying the validity of either φ or $\neg\varphi$. Apart from the formal logic properties of such theories, there is one that links the considerations above to practical machine learning: One accepted measure of learnability is that of *compression*. Hence, the smaller Φ, the more compressed the knowledge. Sadly, this cannot be mapped one-to-one onto the task of finding minimal axiomatisations, because a stronger compressed Φ' may still be more suitable even if $\mathrm{T}h(\Phi) \neq \mathrm{T}h(\Phi)'$.

Learning across means to learn dependencies Using *local* knowledge "in" variables is not sufficient for learning about *dependencies bewtween variables*. Therefore learning about the semantics of a function expressing the relationship between several variables' values requires more. Consider the observations

$$\{\langle\langle 2,4\rangle, 1\rangle, \langle\langle 3,1\rangle, 0\rangle, \langle\langle 2,5\rangle, 0\rangle, \langle\langle 3,6\rangle, 1\rangle\}$$

as instances for snapshots $\langle \vec{x}, \vec{y} \rangle$ of a function f. The hypothesis

$$f((x_0, x_1)) = \begin{cases} 1, & \text{if } x_1 = 2x_0 \\ 0, & \text{else} \end{cases}$$

requires more than just the functions delivering the values of involved variables:

$$\begin{aligned} & x_0(t) = v_0 \wedge x_1(t) = v_1 \wedge g(v_0, v_1) \longrightarrow y(t) = 1 \\ \text{and} \quad & x_0(t) = v_0 \wedge x_1(t) = v_1 \wedge \neg g(v_0, v_1) \longrightarrow y(t) = 0. \end{aligned}$$

First, we either need to have a proper definition of g in our vocabulary of the logic we use to express our specification or we must be able to *invent* such a new *predicate*. Second, the latter formula is not a Horn clause as it contains an additional positive literal: the negated predicate $\neg g$ in the premise. Defining a weaker version of negation as failure (which is compatible with not requiring completeness) simply means that our axiomatisation satisfies

$$f((x_0, x_1)) = 1 \notin \mathrm{Cn}(\Phi) \text{ whenever } x_1 \neq 2x_0.$$

As already pointed out, one needs a predicate symbol g with a suitable semantics such as

$$g(x, y) :\Longleftrightarrow 2x = y \Longleftrightarrow 2x \leq y \wedge 2x \geq y$$

[2] In context of binary classification tasks, $\overline{\langle \vec{x}, y \rangle} = \langle \vec{x}, \overline{y} \rangle$; see footnote 1.

which can be expressed equationally and can be assumed to be predefined in our theory. Things are different with *unknown* predicates whose definitions turn out to be recursive:

$$\{\langle\langle 2,4\rangle,1\rangle,\langle\langle 1,3\rangle,1\rangle,\langle\langle 2,5\rangle,0\rangle,\langle\langle 3,6\rangle,0\rangle\}$$

supports the hypothesis $g(x_0, x_0+2)$, or

$$g(x_0,x_1) \Longleftrightarrow (x_0 = 0 \wedge x_1 = s(s(0))) \vee (x_0 = s(v_0) \wedge x_1 = s(v_1) \wedge g(v_0,v_1)).$$

Hence, $g(x_0, x_1)$ is true, if it is included in

$$\mathrm{Cn}\left(\left\{\begin{array}{rcl} x_0 = 0 \wedge x_1 = s(s(0)) & \longrightarrow & g(x_0,x_1), \\ g(v_0,v_1) \wedge x_0 = s(v_0) \wedge x_1 = s(v_1) & \longrightarrow & g(x_0,x_1) \end{array}\right\}\right) \tag{5.2}$$

which, written as a Prolog program, is

$$g(0,s(s(0))). \qquad g(s(X),s(Y)) :- g(X,Y).$$

This is equivalent to the much more compressed theory

$$\{g(x_0,s(s(x_0)))\}$$

given that the value of x_0 is restricted to $\mathbb{N}$. However, correctness of our hypotheses is guaranteed only if there are no other clauses with a head literal g of the same arity in our theory, as this might allow to infer further, incorrect, instances. What appears to be just a minor issue will turn out to be a challenge when considering concurrent theory construction where parts of a specification of a module are defined in separate and independent theories by mutually ignorant inductive processes.

5.4.2 Learning about dependencies between modules

Information flow along predicates by unification of variables. Assume three modules f, g, h and three logic programs P, Q, R as their respective specifications and predicates p, q, r implementing the functions as logic programs:[3]

$$\text{If } f(\vec{x}) = \vec{y} \text{ then } p(\vec{x}\vec{y}) \text{ is provable}$$

that is, $P \cup \{\neg p(\vec{x}\vec{v})\}$ (where $\vec{v}$ is a sequence of free variables) leads to a contradiction with an instantiation $\vec{y}$ for $\vec{v}$.

[3]The uppercase letters P, Q, R denote Horn theories with arbitrary (additional) clauses; for example definitions of other auxilliary predicates. The lower case predicates p, q, r can be considered the interfaces or C-like `main`-routines. As a conrete example, the theory *Reverse* would export a predicate *reverse* and might include the additional definition of *append*.

Let us further assume that in our system S, g is executed on values computed by f only; i.e. then instantiation of $\vec{x}$ in

$$q(\vec{x}\vec{y}) \tag{5.3}$$

is entirely determined by the variable bindings of $\vec{v}$.[4] Then, the clause

$$h(\vec{x}\vec{w}) : -p(\vec{x}\vec{U}), q(\vec{y}\vec{v})$$

can be proved if

1. $p(\vec{x}\vec{u})$ can be proved with an assignment $\vec{y}$ for $\vec{u}$,
2. $q(\vec{y}\vec{v})$ can be proved with an assignment $\vec{z}$ for $\vec{v}$,
3. and $\vec{z}$ unifies with $\vec{w}$.

Refinement, reordering and abstraction of logic programs. Reverting the argument from above, one changes to an inductive point of view: Consider a sample $\mathbf{s} = \left\{\left\langle \vec{e}, \vec{t}\right\rangle_i : i \in \mathbf{n}\right\}$. Supposing a suitable learning method, a possible result would be

$$h(\vec{a}\vec{w}) : -A, p(\vec{x}\vec{u}), B, q(\vec{y}\vec{v}), C. \tag{5.4}$$

where A, B, C are sequences of predicates such that A determines $\vec{x}$ using values $\vec{a}$; B together with p and A determines $\vec{y}$; A, p, B, q together determine $\vec{v}$; and A, p, B, q, C finally determines $\vec{w}$.
B can be moved inside the definition of p or q transforming the hypothesis to

$$h(\vec{a}\vec{w}) : -A, p_B(\vec{x}\vec{u}_B), q(\vec{y}\vec{v}), C. \tag{5.5}$$

such that A and C can be considered pre- and postconditions that p_B, q needs to satisfy in order to describe h.[5]
Most of the program transformation operators described informally so far, have been described to greater detail as refinement operators in inductive logic programming. For example, the concept of determinacy (equation (5.3)) has been dealt with extensively in [54] and then in [70] to express the amount of information carried along a certain variable. The fact that B as in equation (5.5) can also be moved to the *outside* of p_B to yielding the clause in equation (5.4) forms the basis for the so-called *intra-construction* refinement operator that additionally includes a generalisation operator on the definition of B which has been introduced in [53] already.

[4] Note that the attribute "determinate" differs from the attribute "deterministic": variables $\vec{y}$ are determined by q and $\vec{x}$ if it delivers a possible instantiation for $\vec{y}$ but not that this instantiation is necessarily the only one.

[5] Proving $\{\!|\, A \,|\!\}\, p_B, q\, \{\!|\, C \,|\!\}$ requires the procedural semantics $p_B \S q$ of the declarative program p_B, q.

The biggest problem remains that of identifying the subsets of *relevant* variables carrying the important information. Apart from determinacy measures, usual methods are language restrictions or predicate schemes (similar to rely/guarantee-conditions). The connection to the frame rule is evident: If p does not depend on or is not determined by variables U, then

$$\frac{\{\!|\, \varphi(\vec{x}\vec{y}) \,|\!\}\; p \;\{\!|\, \psi(\vec{x}\vec{y}) \,|\!\}}{\{\!|\, \varphi(\vec{x}\vec{y}) * \Gamma(U) \,|\!\}\; p \;\{\!|\, \psi(\vec{x}\vec{y}) * \Gamma(U) \,|\!\}}$$

where φ and ψ describe the pre- and postconditions and Γ is a set of formulae on variables U. The disjointness condition of free variables in U and the set of variables touched by p is implemented by the notion of determinacy or dependency.

A higher-level logical analysis of induction. Equations (5.1) and (5.2) already capture the idea behind theory induction quite well. The usual notion (in logic machine learning) would be that given a background theory Φ, a set E^+ of theorems one has to be able to prove and one set E^- one must be able to disprove, the task is to find the "*best*" theory H such that

$$\Phi \cup H \models E^+ \quad \text{and } \Phi \cup H \models \sim E^- \tag{5.6}$$

where $\sim E^- = \{\neg\varphi : \varphi \in E^-\}$. This is usually weakened to

$$\Phi \cup H \vdash E^+ \quad \text{and } \Phi \cup H \nvdash E^- \tag{5.7}$$

because of the intractability of semantic entailment and the fact that negations of Horn clauses are not Horn any more.[6] It remains to define which H is *better* than another H'. Again, checking entailment $H \models H'$or vice versa is, in general out of the question. Cheaper versions are syntactical measures, compression or description length measures or accuracy/coverage tests on validation samples. If $\sqsubseteq$ denotes such a partial order between theories that preserves entailment, and the declarative semantics of a logic program interprets the comma "," (i.e. $\fatsemi$) as ordinary conjunction $\wedge$, one can define extremal versions of H by using residual operations:

$$H = \Phi \backslash\!\backslash E^+ \tag{5.8}$$

means that H is the $\sqsubseteq$-maximal set satisfying the left part of equation (5.7). In Heyting-Algebras, this coincides with the E^+-relative pseudocomplement to Φ. At the same time, we want H to generate a largest deductive closure not containing any element of E^-:

$$H = \Phi \backslash\!\backslash \sim E^- \text{ or } H = E^- /\!\!/ \tilde{\Phi} \tag{5.9}$$

where $\tilde{\Phi}$ denotes non-derivability from Φ ($\psi \in \mathrm{Cn}(\tilde{\Phi}) :\Longleftrightarrow \psi \notin \mathrm{Cn}(\Phi)$).

[6] Unless they are unary clauses; i.e. facts. Hence, many if not most approaches require the elements of a sample to be factual although this need not be the case for the theory behind program refinement.

Also, it appears quite natural to postulate

$$\mathrm{Cn}(E^+) \subseteq \mathrm{Cn}(\sim E^-) \text{ and } \mathrm{Cn}(E^-) \subseteq \mathrm{Cn}(\sim E^+) \tag{5.10}$$

Three further paradigms deserve a deeper investigation at this point: Logic programs have both a declarative and a procedural semantics, where the former one corresponds to a test. Therefore, examples E^+ and E^- are tests that must or must not be satisfied when adding H to Φ and logic programs that together with Φ are consistent or inconsistent with H. A hypothesis H is a test, specification and implementation w.r.t. Φ and examples leading to $\top$ or $\bot$ respectively. Finally, Φ can be considered a partial specification. The second paradigm is based on the notion of Hoare triples that, omitting the requirement of extremal weakness or strength, allows relating Φ, H and examples as follows:

$$\{\!|\, \Phi \,|\!\}\, H \,\{\!|\, E^+ \,|\!\} \qquad \{\!|\, H \,|\!\}\, \Phi \,\{\!|\, E^+ \,|\!\} \qquad \{\!|\, \Phi \cup H \,|\!\} \sim E^+ \,\{\!|\, \mathsf{false} \,|\!\} \tag{5.11}$$

Finally, weakest preconditions are well known as box and diamond operators; i.e.

$$\langle E^+ |\, \Phi \leq H \tag{5.12}$$

which is also motivated by the residual based representation in equations (5.8,5.9).[7]

5.4.3 Distributed Learning

Ensemble learning methods like boosting and bagging, [79, 8], were originally developed in the context of statistical or entropy-based learning methods. [57] presents an interpretation in the context of modal logics and relational knowledge discovery.
In this short and concluding section we motivate a view on knowledge discovery as a distributed process itself but restrict ourselves to a bagging-like distribution.
Bagging is a method that relies on splitting the sample data and then aggregates the hypotheses induced on each of them. A partitioning of snapshots (i.e. its division into disjoint subsets) appears a natural first step, but is anything but a wise choice: The hypotheses induced for every such partition are likely to be overfit on the entire sample. Things are even worse, if a partition is taken to be a set of positive instances and the union of all other partitions as negative examples. Hence, overlaps in the sense of deliberate noise (see introductory paragraph of section 5.3) are required. A suitable definition of such required overlaps is an open question though. The next issue in bagging is the definition of a potent aggregation function. With hypotheses being partial theories

$$H_i = \Phi \backslash\backslash E_i^+ \text{ with } i \in \mathbf{n} \tag{5.13}$$

there are no efficiently computable operations $\oplus$, $\otimes$ or $\oslash$ known solving the problems

$$H = \Phi \backslash\backslash E^+ \;=\; (\,\Phi \backslash\backslash E_0^+\,) \oplus (\,\Phi \backslash\backslash E_1^+\,) \oplus \cdots \oplus (\,\Phi \backslash\backslash E_{n-1}^+\,) \tag{5.14}$$

$$H = \Phi \backslash\backslash E^+ \;=\; \Phi \backslash\backslash (E_0^+ \otimes E_1^+ \otimes \cdots \otimes E_{n-1}^+) \tag{5.15}$$

[7] Aside, box and diamond operators motivate a modal extension of logic programs (see, e.g. [64]).

for a given decomposition of E^+, or

$$\Phi \oslash E^+ = \left\{ E_i^+ : i \in \mathbf{n} \right\} \tag{5.16}$$

to compute such a decomposition to satisfy equations (5.14, 5.15).

Chapter 6

Conclusion

In this memo, we gave a broad yet shallow overview of several disciplines sharing common, open research questions:

1. How can autonomous, distributed systems be observed in order to draw conclusions about their components' or entire system behavior?
2. What is needed for a satisfactory theory of distributed, concurrent processes with shared memory?
3. Can we, by way of observation as in 1., induce a concrete theory in terms of 2., that specifies a given system and can we even detect possible reasons for failure?

We also collected and pointed out connections between previous and ongoing work such that the questions above can be specified to the following, tentative, programme for further research:

1. The preliminary relational formalisation of ACMEs needs to be completed.
2. ACMEs can be implemented at several levels of detail (see section 5.2.1) and in several settings:
   (a) Available simulation environments (PARIA) can be used; a very simple but versatile simulation environment would be to use simple shell scripts as described in section 3.1.
   (b) Small autonomous robots implementing behaviour as in section 3.2.3 would provide a bridge between the real world and purely theoretical project work
   (c) Two further thinkable applications are possible in co-operation with PLC- and industrial robot suppliers and user companies.

In a next step, the then-relation algebraic translation of ACMEs shall be adjusted so as to fit in the CKA-based description models of concurrency.
A large portion of the work will be concerned with data collection:

1. Implementations of ACMEs are needed to gather real snapshots and test different techniques of snapshotting.

2. The many different goals in system specification knowledge discovery require sophisticated sampling and snapshotting:

   (a) Can we use results from TLA (see section 4.1.3) to formulate the needed requirements on the snapshots?

   (b) Are they compatible and expressible in CKA?

3. Also, in anticipation of subsequent goals, we shall examine whether these requirements can be formalised in separation logic.

All approaches so far aim at describing concurrent processes and the conditions under which they can safely live together in a common environment. However, they all have a slightly different flavour depending on what their primary goal is: some are focused on states, others on actions; some on control flow, others on data flow; some message passing along channels, others on concurrent memory access. Hence, the work on any of these tasks should be closely connected to ongoing research concerning unified theories of concurrent processes.
This also includes the third and last block of topics. Chapter 5 gives a detailed description of many concrete, open questions; most of them motivated by the question how to learn about a system once its specification is unknown. We therefore only briefly recollect the most important questions without going into much detail since they depend on the findings expected from the work on the questions listed above.

1. Learning specifications of programs with known I/O-behaviour has been dealt with for decades. The first question here is: How can we determine suitable (spatial) parts of a set of snapshots that suffice to describe a single module's behavior?

2. Given large snapshots or sets of (overlapping) spatial parts of snapshots, how can we identify interfaces and groups of semantically grouped modules? — How does this relate to the notion of interfaces of concurrent processes and refinement of sets of processes?

3. Snapshots take a certain "exposure" time during which data may travel through space. As we focus on learning declarative hypotheses, time is a source of noise. On the other hand, if snapshots are collected by the distributed processes themselves, their aggregation is inherently inconsistent. How can we ensure time- or memory-space consistent data to learn from?

4. Dependency between processes is easy to detect if their specifications reveal that one passes to another. How can we detect the dependency between processes without knowing which memory region represents an interface between them?

Finally, it would be interesting to optionally investigate how the methods for knowledge discovery (that up to know were implicitely taken to be fixed) themselves can be improved.

Bibliography

[1] Anand S. Rao, M. P. G. Modeling Rational Agents within a BDI-Architecture. In *Proc. 2nd Intl. Conf. on Principles of Knowledge Representation and Reasoning (KR'91)* (1991).

[2] Baier, C., and Katoen, J.-P. *Principles of Model Checking*. MIT Press, 2008.

[3] Ben-Hur, A., Ong, C. S., Sonnenburg, S., Schölkopf, B., and Rätsch, G. Support Vector Machines and Kernels for Computational Biology. *PLoS Computational Biology 4*, 10 (October 2008).

[4] Blum, A., and Mitchell, T. Combining Labeled and Unlabeled Data with Co-Training. In *COLT' 98 Proceedings of the eleventh annual conference on Computational learning theory* (1998), ACM, pp. 92–100.

[5] Bolton. *Programmable Logic Controllers*, 4 ed. Newnes, 2006, ch. 11. Ladder and Functional Block Programming, pp. 454–.

[6] Braitenberg, V. *Vehicles: Experiments in Synthetic Psychology*. The MIT Press, 1986.

[7] Bratman, M. E. *Intention, Plans, and Practical Reason*. CSLI - Stanford Publications, 1999. Reprint of 1987.

[8] Breiman, L. Bagging Predictors. *Machine Learning 7*, 2 (August 1996), 123–140.

[9] Brookes, S. A semantics for concurrent separation logic. *Theoretical Computer Science 375* (2007). Festschrift for John C. Reynolds's 70th Birthday, 2007. Extended version of paper from CONCUR 15.

[10] Brooks, R. A. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation 2*, 1 (March 1986), 14–23.

[11] Brooks, R. A. Intelligence without representation. *Artificial Intelligence 47* (1987).

[12] Brooks, R. A. Intelligence Without Reason. Tech. rep., MIT AI Laboratory, 1991.

[13] Carver, N., and Lesser, V. Evolution of blackboard control architectures. *Expert Systems with Applications 7*, 1 (1994).

[14] Corkill, D. D. Blackboard Systems. *AI Expert 6*, 9 (1991).

[15] Douglas, A., Wood, A., and Rowstron, A. Linda implementation revisited. In *Proc. of the 18th World Occam and Transputer User Group* (1995), IOS Press, pp. 125–138.

[16] Enderton, H. B. *A mathematical introduction to logic*, 2nd ed. Academic Press, 2001.

[17] Gelernter, D. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems 7*, 1 (1985), 80–112.

[18] Glück, R. L. J., and Krebs, F. Towards Interactive Verification of PLC Programs using MKA and KIV. In Kahl et al. [40].

[19] Goldberg, D. Genetic and evolutionary algorithms come of age. *Communications of the ACM 37*, 3 (1994), 113–119.

[20] Gulwani, S., Hernandez-Orallo, J., Kitzelmann, E., Muggleton, S. H., Schmid, U., and Zorn, B. Inductive Programming Meets the Real World. *Communications of the ACM 58*, 11 (November 2015), 90–101.

[21] Hardin, C. How the Location of * Influences Complexity in Kleene Algebra with Tests. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2004)* (2005), F. Baader and A. Voronkov, Eds., vol. 3452 of *LNCS*, Springer.

[22] Haubrich, T. Simulation and Visualisation of Swarms and Flocks (orig. German). Master's thesis, Bonn-Rhein-Sieg University of Applied Sciences (BRSU), 2011.

[23] Hayes, P. J. The Frame Problem and Related Problems in Artificial Intelligence. n 0, Stanford University, 0, 0 1971.

[24] Hayes-Roth, B. A blackboard architecture for control. *Artificial Intelligence 26*, 3 (1985).

[25] Hermann, M., Hentek, T., and Otto, B. Design Principles for Industrie 4.0 Scenarios: A Literature Review. Tech. Rep. 01/2015, Faculty of Engineering, Technical University Dortmund, 2015.

[26] Hielscher, F. Entwicklung einer Roboterplattform für kognitive, autonome Systeme. Master's thesis, Universität Augsburg, 2008.

[27] Hoare, C. A. R. Communicating Sequential Processes. *Communications of the ACM 21*, 8 (1978), 666–667.

[28] Hoare, C. A. R. A Theory of Programming: Denotational, Algebraic and Operational Semantics. November 1999.

[29] Hoare, C. A. R. *Communicating Sequential Processes*. 2004. First published as [27].

[30] Hoare, C. A. R., Möller, B., Müller, M. E., and Struth, G. Unifying Models and Laws for Concurrency and Distribution. In *The 6th International Symposium on Unifying Theories of Programming* ((in preparation)), LNCS, Springer.

[31] Hoare, C. A. R., Möller, B., Struth, G., and Wehrman, I. Concurrent Kleene Algebra. Tech. Rep. 2009-04, Institut für Informatik, Universität Augsburg, 2009.

[32] Hoare, C. A. R., Möller, B., Struth, G., and Wehrman, I. Concurrent Kleene Algebra and its Foundations. *The Journal of Logic and Algebraic Programming 80*, 6 (2011), 266–296.

[33] Hoare, T., Möller, B., Struth, G., and Wehrman, I. Concurrent Kleene Algebra. In *CONCUR 2009 - Concurrency Theory* (2009), vol. 5710 of *LNCS*, pp. 399–414.

[34] Hoare, T., Möller, B., Struth, G., and Wehrman, I. Foundations of Concurrent Kleene Algebra. In *Relations and Kleene Algebra in Computer Science* (2009), R. Berghammer, B. Möller, and A. Jaoua, Eds., vol. 5827 of *LNCS*, Springer. Prelim. version in: Universität Augsburg Technical Report 2009-5, Institute of Computer Science, University of Augsburg, 2009.

[35] Hoare, T., von Staaden, S., Möller, B., Struth, G., Villard, J., Zhu, H., and O'Hearn, P. Developments in Concurrent Kleene Algebra. In *Relational and Algebraic Methods in Computer Science* (2014), P. Höfner, P. Jipsen, W. Kahl, and M. E. Müller, Eds., vol. 8428 of *LNCS*, Springer, pp. 1–18.

[36] Jipsen, P. Concurrent Kleene Algebra with Tests. In *Relational and Algebraic Methods in Computer Science*, P. Höfner, P. Jipsen, W. Kahl, and M. E. Müller, Eds., vol. 8428 of *Lecture Notes in Computer Science*. Springer, 2014, pp. 37–48.

[37] Jobst, S. Eine Plattform zur Realisierung von Multiagentensystemen mit mobilen Robotern. Master's thesis, Universität Augsburg, 2007.

[38] Jobst, S. Hard- und Softwareunterstützung für modulare Agenten. Master's thesis, Universität Augsburg, Augsburg, 2009.

[39] Kagermann, H., Wahlster, W., and Helbig, J. Recommendations for Implementing the Strategic Initiative "Industrie 4.0": Final report of the Industrie 4.0 Working Group. Tech. rep., Bundesministerium für Wirtschaft und Energie, 2013.

[40] Kahl, W., Winter, M., and Oliveira, J., Eds. *Relational and Algebraic Methods in Computer Science (15th Intl. Conf. RAMiCS 2015)* (October 2015), no. 9348 in Lecture Notes in Computer Science (LNCS), Springer.

[41] Kleene, S. C. Representation of events in nerve nets and finite automata. Tech. Rep. RM-704, USAF RAND, RAND Corp, 1700 Main St, Santa Monica, CA, December 1951.

[42] Krebs, F. Entwicklung eines Systems zur Implementierung von Layered Module Architekturen. Master's thesis, Universität Augsburg, 2008.

[43] Lamport, L. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (1993), 872–923.

[44] Lamport, L., and Chandy, M. Distributed Snapshots: Determining Global States of a Distributed System. *ACM Transactions on Computer Systems 3*, 1 (February 1985), 63–75.

[45] Lau, T. Why Programming-By-Demonstration systems fail: Lessons learned for usable AI. *AI Magazine 30*, 4 (2009).

[46] Li, J., Zheng, S., Chen, B., Butte, A. J., Swamidass, S. J., and ZhiyongLu. A survey of current trends in computational drug repositioning. *Briefings in Bioinformatics Advance Access* (March 2015).

[47] Markus Müller-Olm, David Schmidt, B. S. Model-Checking: A Tutorial Introduction. In *Proc. 6th Static Analysis Symposium* (1999), G. File and A. Cortes, Eds., vol. 1694 of *LNCS*, pp. 330–354.

[48] Maturana, H. R., and Varela, F. J. *Tree of Knowledge: Biological Roots of Human Understanding*. Shambala, 1987.

[49] McCarthy, J., and Hayes, P. J. Some philosophical Problems from the standpoint of Artificial Intelligence. *Machine Intelligence 4* (1969).

[50] McCulloch, W. S., and Pitts, W. A Logical Calculus of the Ideas immanent in Nervous Activitiy. *Bulletin of Mathematical Biophysics 5*, 4 (December 1943), 115–133.

[51] Michalski, R. A theory and methodology of inductive learning. In *Machine Learning: An Artificial Intelligence Approach*, R. Michalski, J. Carbonnel, and T. Mitchell, Eds. Tioga, Palo Alto, CA, 1983, pp. 83–134.

[52] Mitchell, T. *Machine Learning*. McGraw-Hill, New York, 1997.

[53] Muggleton, S., and Buntine, W. Machine invention of first-order predicates by inverting resolution. In *Proc. 5th Intl. Conf. on Machine Learning* (1988), Kaufmann, pp. 339–352.

[54] Muggleton, S., and Feng, C. Efficient induction of logic programs. In *Proc. 1st Conf. on Algorithmic Learning Theory* (Tokyo, 1990), Ohmsha, pp. 368–381.

[55] Muggleton, S., and Marginean, F. Logic-based Machine Learning. In *Logic-Based Artificial Intelligence*, J. Minker, Ed. Kluwer, 2000, pp. 315–330.

[56] Müller, M. E. Learning for User Adaptive Systems: Likely Pitfalls and Daring Rescue. In *Proc. 11th National Workshop on Adaptivity and User Modeling in interactive Software Systems (ABIS-2003)* (2003), pp. 323–326.

[57] Müller, M. E. Modalities, Relations, and Learning. In *Relations and Kleene Algebra in Computer Science*, R. Berghammer, B. Möller, and A. Jaoua, Eds., vol. 5827 of *LNCS*. Springer, 2009, pp. 260–275.

[58] Müller, M. E. *Relational Knowledge Discovery*. Cambridge University Press, 2012.

[59] Müller, M. E. Roughness by Residuals. In *Relational and Algebraic Methods in Computer Science (RAMiCS'15)* (2015), W. Kahl, M. Winter, and J. N. Oliveira, Eds., no. 9348 in LNCS, Springer.

[60] Müller, M. E., Häring, M., Hielscher, F., Jobst, S., Krebs, F., and Prüller, A. PARIA - Abschlußbericht. Tech. rep., Universität Augsburg (in preparation), 2009.

[61] Müller, M. E., Krebs, F., and Hielscher, F. Relational Cognitive Structures for Intelligent Agent and Robot Control. In *Systems, Man, and Cybernetics (SMC-2008)* (2008), IEEE.

[62] Müller, M. E., and Thosar, M. Learning to Understand by Evolving Theories. In *KRR Workshop at ICLP 2013* (2013).

[63] Nguyen, H. D., Yoshihara, I., Yamamori, K., and Yasunaga, M. Aligning Multiple Protein Sequences by Parallel Hybrid Genetic Algorithm. *Genome Informations 13* (2002).

[64] Nguyen, L. A. Multimodal logic programming. *Theoretical Computer Science 360*, 1 (2006).

[65] O'Hearn, P. W., Petersen, R. L., Villard, J., and Hussin, A. On the relation between Concurrent Separation Logic and Concurrent Kleene Algebra. *Journal of Logical and Algebraic Methods in Programming*, Special Issue "RAMiCS 2014" (2014).

[66] O'Hearn, P. W., Yang, H., and Reynolds, J. C. Separation and Information Hiding. *ACM Transactions on Programming Languages and Systems 31*, 3 (April 2009).

[67] Oliveira, J. N. A relation-algebraic approach to the "Hoare logic" of functional dependencies. *Journal of Logical and Algebraic Methods in Programming 83* (2014), 249–262.

[68] Pavlovic, O., Pinger, R., and Kollmann, M. Automated Formal Verification of PLC Programs Written in IL. In *Proceedings of 4th International Verification Workshop VERIFY'07 (in connection with CADE-21)* (2007), CEUR-WS.org.

[69] Pratt, V. R. Action Logic and pure induction. In *Logics in AI (JELIA '90)* (1991), J. van Eijck, Ed., vol. 478 of *LNAI*, Springer, pp. 92–120.

[70] Quinlan, J. Determinate literals in inductive logic programming. In *Proc. 12th Intl. Joint Conf. on Artificial Intelligence* (San Mateo, CA:, 1991), Morgan-Kaufmann, pp. 746–750.

[71] Raedt, L. D. *Logical and Relational Learning.* Cognitive Technologies. Springer, 2008.

[72] Rao, A. S., and Georgeff, M. P. BDI Agents: From Theory to Practice. Tech. Rep. 56, Australian Artificial Intelligence Institute, 1995.

[73] Reif, W., Schellhorn, G., Stenzel, K., and Balser, M. *Automated Deduction — A Basis for Applications*, vol. 9 of *Applied Logic Series.* Springer, 1998, ch. Structured Specifications and Interactive Proofs with KIV, pp. 13–39.

[74] Reiter, R. *Readings in nonmonotonic reasoning.* Morgan Kaufmann, 1987, ch. On closed world data bases On closed world data bases On closed world data bases, pp. 300–310.

[75] Reynolds, J. C. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE Symposium on Logic in Computer Science* (2002).

[76] Rosenblatt, F. The Perceptron: A probabilistic model for information storage and organisation in the brain. *Psychological Reviel 65*, 6 (1958).

[77] Rowstron, A., and Wood, A. An efficient distributed tuple space implementation for networks of heterogenous workstations. (online; publication details unknown), 1996.

[78] Rumelhart, D. E., and McClelland, J. L. *Parallel Distributed Processing - Vol. 1: Foundations*, vol. 1. MIT Press, 1986.

[79] Schapire, R. E. The Boosting Approach to Machine Learning: An Overview. MSRI Workshop on Nonlinear Estimation and Classification, 2002.

[80] Schlich, B., Brauer, J., Wernerus, J., and Kowalewski, S. Direct Model Checking of PLC Programs in IL. In *2nd IFAC Workshop on Dependable Control of Discrete Systems* (2009), vol. 2 of *Dependable Control of Discrete Systems*, International Federation of Automatic Control.

[81] Schmidt, D. A. *Denotational semantics: a methodology for language development.* Allyn and Bacon, 1986.

[82] Schmidt, D. A. Programming Language Semantics. Tech. rep., Kansas State University, 1995.

[83] Siebertz, F. PARIA-Roboter Software Manual. Tech. rep., University of Applied Sciences Bonn-Rhein-Sieg, 2010.

[84] Soliman, D., and Frey, G. Verification and Validation of Safety Applications based on PLCopen Safety Function Blocks using Timed Automata in Uppaal. In *2nd IFAC Workshop on Dependable Control of Discrete Systems* (2009), Dependable Control of Discrete Systems, International Federation of Automatic Control.

[85] Sonnenburg, S., Zien, A., Philips, P., and Rätsch, G. POIMS: Positional Oligomer Importance Matrices - Understanding Support Vector Machines Based Signal Detectors. *Bioinformatics 24*, 13 (2008).

[86] Sun, R., and Giles, C. L., Eds. *Sequence Learning.* No. 1828 in Lecture Notes in Artificial Intelligence. Springer, 2000.

[87] Tennent, R. D. The denotational semantics of programming languages. *Communications of the ACM 19*, 8 (August 1976), 437–453.

[88] Wehrman, I., Hoare, C. A. R., and O'Hearn, P. W. Graphical Models of Separation Logic. *Information Processing Letters 109*, 17 (2009), 1001–1004.

[89] Weissmann, M., Bedenk, S., Buckl, C., and Knoll1, A. Model Checking Industrial Robot Systems (18th International SPIN Workshop). In *Model Checking Software*, A. Groce and M. Musuvathi, Eds., vol. 6823 of *LNCS*. Springer, 2011, pp. 161–176.

[90] Wooldrigde, M. *Introduction to Multi Agent Systems.* Wiley and Sons, 2002.

[91] Wooldrigde, M., and Jennings, N. R. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review 10*, 2 (1995), 115–152.