JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche Fakultät

Semantic Assistance for Industrial Automation Based on Contracts and Verification

DISSERTATION

zur Erlangung des akademischen Grades

Doktor

im Doktoratsstudium der

Technischen Wissenschaften

Eingereicht von:
DI (FH) Dominik Hurnaus

Angefertigt am:
Institut für Systemsoftware

Beurteilung:
Prof. Dr. Dr. h.c. Hanspeter Mössenböck (Betreuung)
Prof. Dr. Armin Biere

Mitwirkung:
Dipl.-Ing. Dr. Herbert Prähofer

Linz, Oktober 2009

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, Oktober 2009 DI (FH) Dominik Hurnaus

Acknowledgment

I wish to thank all those who helped me. Without them, I could not have completed this thesis. First and foremost, I thank my advisor Prof. Hanspeter Mössenböck and Dr. Herbert Prähofer for their support and comments on ideas, papers and this thesis.

I am particular grateful to Dr. Herbert Prähofer for commenting on drafts of the thesis and enhancing the quality of this work. Without his helpful comments and discussions this work would not have been possible. Likewise I thank Prof. Armin Biere for taking the responsibility of the second thesis advisor and dissertation committee member.

Special thanks go to my friends and colleagues DI Markus Löberbauer, DI Roland Schatz, DI Christian Wirth, and DI Reinhard Wolfinger for the fruitful discussions which helped to develop ideas put forward in the thesis. Likewise I thank Johannes *Gasi* Straßmayr for his contributions to the program visualization tool and for helping carrying out the evaluation studies.

This work has been conducted in the module *"Domain-Specific Languages for Industrial Automation"* at the Christian Doppler Laboratory for Automated Software Engineering in cooperation with KEBA AG. Therefore, I want to thank KEBA AG and the Christian Doppler Forschungsgesellschaft for funding the project and our contact persons at KEBA AG, Dr. Ernst Steller, Dr. Michael Garstenauer, and DI Gottfried Schmidleitner for their continuous support.

My deepest and sincere thanks go to my girlfriend Juliane for her love and patience. She continuously encouraged me to do my best and she was always there when I needed her most. Finally, I would like to thank everyone who contributed to this thesis and especially my family for their great support during the years of study.

Kurzfassung

In der Industrieautomation müssen Endbenutzer oft kleinere Änderungen an Steuerungsprogrammen der Maschinen vornehmen. Diese Endbenutzer sind meist Maschinenbediener, die wenig bis gar keine Programmierkompetenz haben. Dennoch müssen sie in sicherheitskritsche Steuerungsprogramme eingreifen, bei denen Testläufe nicht möglich sind.

Der in dieser Arbeit beschriebene Ansatz wird basiert auf *Verifikation* von Steuerungsprogrammen. Mittels Verifikation wird bewiesen, dass ein Softwaresystem bestimmte Eigenschaften in jeder möglichen Ausführung einhält. Für die Verifikation von Software ist es notwendig, die gewünschten Eigenschaften der Software in Kontrakten zu beschreiben. Die Kontrakte, die in dieser Arbeit verwendet werden, beschreiben gültige Aufruffolgen und Einschränkungen.

Semantic Assistance - ein neues Konzept, das in dieser Arbeit vorgestellt wird - verwendet die Ergebnisse der Verifikation, um Endbenutzern bei der Programmierung zu helfen. Diese Hilfe umfasst interaktive Unterstützung bei Programmänderungen, Vorschläge gültiger Programmteile sowie Visualisierung von Zuständen von Maschinenkomponenten. Im Falle einer Verletzung der Kontrakte können automatische Programmänderungen vorgeschlagen werden, die die Programmfehler korrigieren.

Verifikation und Semantic Assistance wurden in die Entwicklungsumgebung der domänenspezifischen Sprache MONACO integriert. Fallstudien zeigen, dass der Ansatz von Kontrakten und Semantic Assistance praktikabel ist. Darüber hinaus wurde festgestellt, dass Einschränkungen auf MONACO Systemen unkompliziert gefunden werden können und die statische Überprüfung dieser Einschränkungen die Laufzeitressource der Steuerungshardware entlasten.

Abstract

In the field of industrial automation end users often have the task of making changes and small adaptations to control programs of their machines. These end users (machine operators) usually lack software engineering expertise, yet they have to intervene in safety-critical, highly dependable systems where it is not possible to run any offline tests.

Verification is used to proof that specific properties of software systems hold in every possible execution of the system. This is in contrast to testing, which can only show that a property holds in a given situation with a defined input. For software verification it is necessary to formally describe these properties in contracts, containing possible call sequences and constraints on system states. Information of the intermediate steps of the verification process are stored with the software implementation to be reused later.

Semantic Assistance - a new concept introduced in this thesis - uses the results of a verification process to give guidance to end-user programmers. This guidance ranges from interactive assistance on valid routine calls to visualization of program states in form of a schematic view of the machine. In case of a contract violation, it is possible to automatically generate program repair proposals to eliminate the violation.

Verification and Semantic Assistance are integrated into the MONACO IDE, a system for creating control programs with the domain-specific language MONACO. Case studies and evaluation results show that this approach is feasible for different types of control programs. Furthermore, we experienced that finding constraints of systems is uncomplex and checking these constraints statically removes substantial runtime overhead.

Contents

Chapter 1

Introduction

This thesis presents concepts and tools supporting end-user programming of industrial automation solutions. In industrial automation the end users, which can be domain experts or less experienced operators at a machine, often have to make changes to the control programs of their machine automation solutions. Those people — while they need to intervene in safety-critical systems — usually lack software engineering expertiese. Moreover, they often have to modify programs on a running machine and make those changes effective without a chance to run offline tests or try the changed program in a test environment.

We have observed that in such a setting constraints on the operations as well as dependencies between machine components apply in an obvious and natural way. Those are constraints on valid sequences of operations of components and inter-dependencies between operations of components. Instead of having these tacit assumptions reside in the minds of end-user programmers, they should be formalized and used to constrain end-user programmers so violations cannot occur in the first place.

The work presented in this thesis adopts techniques from formal interface specification [dAH01, Mey86], model checking [CGP99], and artificial intelligence [KM91] to make this support possible. Formal interface specification techniques are used to specify the sequencing constraints of component calls, knowledge about state properties of components, as well as inter-component constraints. Model checking and artificial intelligence techniques are then used to verify that a client program obeys these specifications and constraints.

Based on these techniques, we have introduced means to support end users in programming, which we call *Semantic Assistance*. This works similar to code assist techniques (Visual Studio IntelliSense, Eclipse content assist, ...) where programmers get suggestions of syntactically correct method calls based on the current code position. *Semantic Assistance*, however, is based on semantic knowledge represented in component contracts.

1.1 Background and Motivation

The work is based on the domain-specific language MONACO [PHM06, PHWM07, PHS+08a] which is described in Chapter 3 of this thesis.

MONACO (*Mo*deling *N*otation for *A*utomation *Co*ntrol) is a domain-specific language for machine automation programming. It allows programming the reactive part of an automation solution. It therefore has language constructs to express machine operation sequences, has strong support for dealing with exceptional situations and allows parallel activities. The behavioral model of MONACO is close to StateCharts [Har87], although it uses an imperative, Pascal-like style of programming.

The most essential statements in the MONACO language are synchronous routine calls which execute control tasks, **WAIT** statements for implementing wait conditions, and the **PARALLEL** statement used to allow concurrent execution of several activities. Additionally, **ON**-handlers, can be used to implement reactions to exceptional situations.

An important language feature is the component-based approach, i.e., components are modular units which exclusively communicate over defined interfaces. Strict correspondence between the hardware components of the machine and the software components controlling the machine parts is pursued. The interface specifications in MONACO consist of (1) routines which represent the actions and tasks that can be fulfilled by this component, and (2) functions which allow accessing state properties. That means, routines specify how a component can be controlled and functions specify the feedback a component provides.

Moreover, components are arranged in a hierarchical fashion of superordinate and subordinate components which reflects the hierarchical structure of the real machine and accounts for the hierarchical nature of control tasks. Components that are the leaves of the component hierarchy are called native components and are implemented in a native language of the control machine (e.g., C++) to interface with the hardware or lower control layers. Higher up in the hierarchy there are several coordination components which coordinate and supervise the operations of their subcomponents. Chapter 3 presents the language MONACO in more detail.

End-user programming is typically performed at the topmost or higher control layers. End users are presented a so-called "end-user window" which provides a limited view of the control program. Typically, an end user is only allowed to add some functionality, reorder routine calls, add conditional statements, or change some parameter settings.

On the other side, there are constraints and dependencies on the operations of the components, which must be enforced in any program. Although often quite obvious (see Section 8), it is hard or even impossible for end users to follow these constraints while they modify a program. So far, restrictions and constraints are checked in a separate program section. However, the checks are done at runtime, often resulting in emergency stops and expensive machine downtimes. It is therefore highly desirable to have a means of checking and enforcing those constraints and restrictions already at compile time.

1.2 Outline of our Approach

Our approach is based on the specification of dynamic contracts for components, automata simulation, a knowledge deduction process which derives knowledge about program properties at code positions, and assistance techniques which exploit this knowledge. The assistance tools give immediate feedback on contract and constraint violations, generate proposals of valid program changes and present those to the end-user programmer. Additionally, the machine state for a certain location in the code can be visualized at editing time, such that the end-user programmer can get a better understanding of a program.

Figure 1.1 depicts an overview of our approach. First, the valid behavior of the components is described in *protocol contracts* and *constraints* (2),

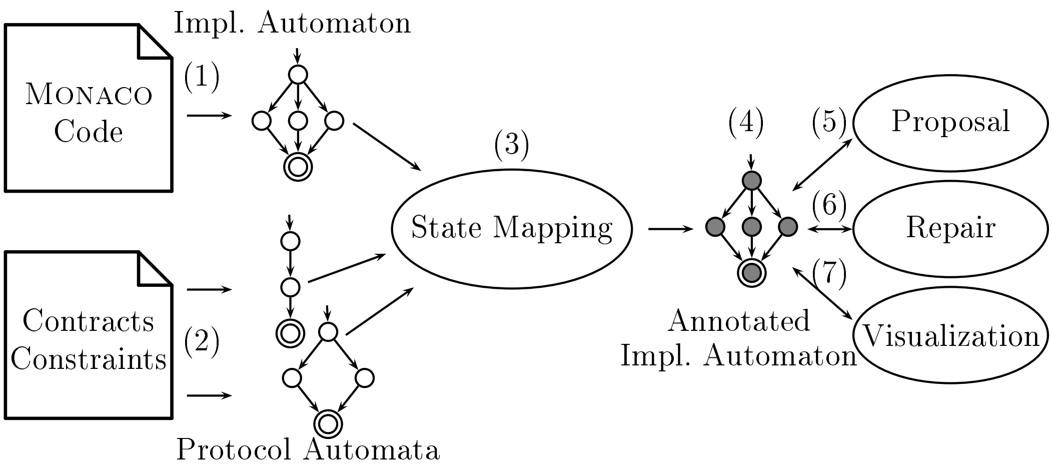


Figure 1.1: Protocol contracts and the state mapping algorithm are the basis for a variety of end-user guidance applications.

which are translated into protocol automata. Second, the behavior of the component implementation is translated into *implementation automata* (1) containing control flow information as well as Boolean conditions affecting the control flow. Next, a state mapping algorithm (3) establishes a weak simulation relation [Bie08] between the implementation automaton and the protocol automata of the subcomponents. It associates state knowledge with states of the automaton and updates this knowledge while checking the implementation for contract violations. The resulting annotated implementation automaton (4) is then used in different end-user support systems as follows:

- The IDE provides immediate feedback about contract and constraint violations at the code position in the editor.
- Valid routine calls (5) to subcomponents are proposed based on the contracts of the subcomponents while observing constraints.
- Semantic program repair (6) gives proposals on how a program violating contracts or constraints can be changed such that the resulting program complies with contracts and constraints.
- Program state visualization (7) uses knowledge generated from the state mapping algorithm to visualize the state of components at a certain location in the code.

1.3 Project History

This work is part of the project MONACO of the Christian Doppler Laboratory for Automated Software Engineering[1] at the Institute for System Software[2] at the Johannes Kepler University, Linz, Austria[3]. The laboratory was founded in February 2006, in cooperation with Keba AG, Austria[4] and is funded by the Christian Doppler Forschungsgesellschaft, Austria[5].

The project started in 2006 with the definition of a first version of the domain-specific language MONACO, a compiler, and a runtime environment [Hur06], [PHM06]. In July 2006 a second version of the runtime environment and a visual programming environment [PHWM07] has been created.

In December 2006, first ideas about MONACO code verification and using contracts to guide end users emerged. We also worked on compilers and runtime environments in C, the integration into the existing runtime of Keba, and on an end-user friendly UI configuration tool based on variability models [PHS+08a], [HW08]. We started first experiments with contracts and the description of the behavior of MONACO components. In late 2007, prototypes of the code verification algorithm existed (yet without pre- and postconditions), in 2008, the missing pre- and postconditions as well as the program repair functionality were implemented [PHS+08c]. In 2009 program visualization support was added [Str09].

1.4 Structure of the Thesis

This thesis is organized as follows: Chapter 2 reviews techniques and tools which serve as background and motivation for our research. Chapter 3 presents the domain-specific language MONACO. Subsequent Chapters 4–6 explain the algorithms and data structures used to abstract from MONACO code, verify it, and generate knowledge. Chapter 7 presents Semantic Assistance tools based on the results of the verification process. The tools are

[1] http://ase.jku.at
[2] http://ssw.jku.at
[3] http://www.jku.at
[4] http://www.keba.at
[5] http://www.cdg.ac.at

used to guide end users. In case of contract violations they help finding valid program repair strategies. A state deduction process is used for a design-time program visualization tool. Case studies in Chapter 8 demonstrate the applicability of the presented approach to realistic problems. Chapter 9 discusses related projects on verification of component-based systems, description of component behavior, program repair, and program visualization. Finally, Chapter 10 concludes the thesis with a summary of the most significant parts and a summary of the contributions.

Chapter 2

State of the Art

"Beware of bugs in the above code;
I have only proved it correct,
not tried it."
- *Donald Knuth*

This chapter provides a brief overview over the state of the art of the topics which form the background of this work. Section 2.1 introduces code completion systems currently available for popular development environments. Section 2.2 reviews formal methods, model checking, and propositional satisfiability. The last section introduces the topic of belief revision and belief update.

2.1 Code Completion

Source code text editors in modern integrated development environments (IDEs) give programmers versatile support in performing their tasks. Besides syntax highlighting, IDEs also provide users with suggestions and information related to the current context. This information is either displayed as an overview over the current context (e.g., the Outline view in the Eclipse IDE) or as syntax-directed code completion proposals that pop up while the programmer types code.

While these popup menus are named differently in their respective IDEs (e.g., *Content Assist* in Eclipse, *IntelliSense* in Microsoft Visual Studio) they all have similar functionality: Proposing valid code (e.g., class members) using meta data (syntax tree), reflection or heuristics based on the current context.

Microsoft IntelliSense

Microsoft®IntelliSense is the code completion facility of Microsoft Visual Studio®. It uses .NET reflection and the introspection facilities of COM to establish a database of symbols and scopes, which is consulted when the user enters code in the editor. Syntactically suitable symbols (class names, method names, field names, variable names, etc.) are then presented in a drop-down box and help to find elements available in the scope of the context.

Eclipse Content Assist

Similar to Microsoft's code completion implementation, the Eclipse JDT (*Java Development Tools*) provide a facility called Content Assist [AL04]. Content Assist takes the guesswork out of coding by helping the programmer to

- find a given type
- find a given field or method of an object
- enter method parameter values

Additionally, Eclipse provides contextual information about the current file in the so-called *Outline* view.

Productivity Tools

For Microsoft Visual Studio there exist many third-party add-ins which enhance the capabilities of the built-in IntelliSense by providing a richer set of heuristics to find the elements that may be needed in a specific context. As an example, JetBrains ReSharper (http://www.jetbrains.com/resharper) provides *advanced code completion* which proposes symbols that, for example, meet the expected type of an assignment.

Shortcomings

All the productivity tools mentioned above provide code completion and code proposals based on the local, syntactic context of the editing position in the code. This context is searched for information about the static program structure consisting of variables and member declarations.

While this locality makes the approaches applicable to a wide variety of scenarios, they fail to take into account state information (*semantic information*) and information about component behavior. For example, after typing a variable name and a dot the tools infer the type of the variable and suggest all methods that can be applied to this variable. However, they fail taking into account whether a suggested method call would be semantically correct at the current position, i.e., whether the call would be legal in the sequence of method calls that is defined by the contract of the variable's type.

2.2 Formal Methods

The term formal methods describes techniques for the specification, synthesis and verification of hardware and software systems. Figure 2.1 shows a big picture of formal methods:

Formal Specification. Formal specification languages abstractly describe what an implementation should do. These descriptions (models) contain information about the states of a system and the operations which cause the system to make transitions to other states. Well-known specification languages are abstract state machines (*ASM*) [GKOT00], the vienna development method specification language (*VDM-SL*) [ISO96a], the Z notation [ASM80], and temporal logics (see Section 2.2.2).

Formal Synthesis. Formal synthesis is the translation of a specification into a more concrete implementation (see Figure 2.2). This step is also referred to as *refinement* or *transformation*. If all translation steps can be proven to be correct, an actual implementation can be generated which is *correct by construction* (i.e. it is correct with respect to the specification).

Formal Verification. Formal verification uses mathematical techniques to ensure that a system conforms to some precisely expressed notion of functional correctness (specification) [Bje05].

Section 2.2.1 will detail on model checking, while Section 2.2.2 introduces formal specification languages. Propositional satisfiability and tools for solving satisfiability problems are presented in Section 2.2.3.

2.2.1 Model Checking

Model checking is an automatic technique for verifying finite state concurrent systems [CGP99]. It is a formal verification method which verifies a certain property of a system by exploring all reachable states of the system. The advantages of model checking over other verification approaches are that it can be applied fully automatically, and if a state has been found where the property is violated, model checking generates a counterexample, i.e., a sequence of transitions that leads the system into the faulty state. This counterexample can then be used to locate the actual fault of the system.

There are two special types of properties that are of interest in model checking:

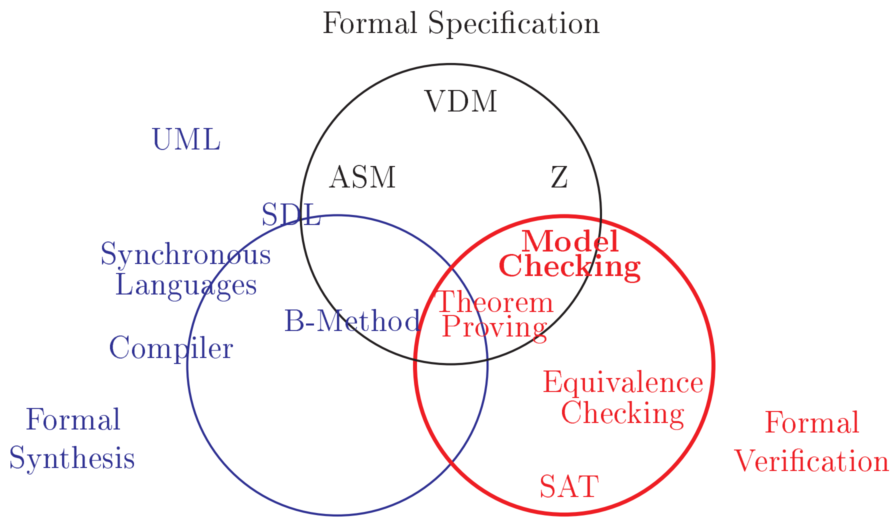


Figure 2.1: Overview over formal methods [Bie08].

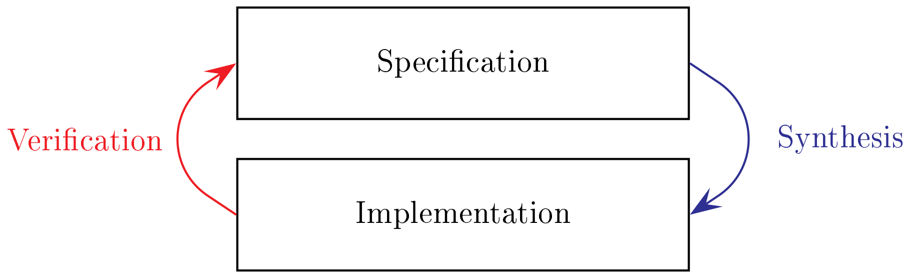


Figure 2.2: Verification and Synthesis.

Safety. Safety properties assert that nothing bad happens. For example: *"As long as the service door is open, the machine must not start"*.

Liveness. Liveness properties assert that some progress eventually happens. For example: *"The traffic light eventually turns green"*.

Model checking tools use these properties encoded in some specification language (see Section 2.2.2) to verify the system. Since model checking tools traverse all reachable states of a system, these states need to be represented in memory. The main problem of model checking is, that large systems often consist of much more states than can be represented in memory. This main problem is therefore called the *state explosion problem*. Many approaches exist to overcome this problem:

Symbolic Model Checking [McM92]. Symbolic model checking avoids building a complete state graph by using formulas to represent sets of states.

Partial Order Reduction [CGP99]. Partial order reduction reduces the size of the state graph by partially expanding local states in a synchronous composition of components.

Compositional Model Checking [BCC98]. Compositional or modular model checking partitions a system into a set of components communicating over simple interfaces. Instead of checking the parallel composition of all components, each component is checked separately, assuming certain behavior of the other components. The validity of these assumptions is later verified when the respective component is checked.

Predicate Abstraction [Das03]. Instead of checking a large system, an abstract model of the system is created. This abstract model does not reflect all properties of the original system, while it still contains enough information to verify the desired correctness properties.

While all of these techniques aim at making model checking feasible, very few tools provide feedback about the checking process other than providing a counterexample trace or reusing the counterexample to further detail the abstraction (counterexample guided abstraction refinement, *CEGAR* [CL00]).

Tools

Model checking tools (*model checkers*) exist for various application areas and various programming languages. The following list shows three prominent model checkers, all based on different languages.

SPIN. SPIN (Simple Promela Interpreter) [Hol03] is a model checker developed by Gerard J. Holzmann and can be used to check verification models specified in *Promela*, a verification modeling language aimed at modeling the behavior of concurrently executing processes[1].

BLAST. BLAST [BHJM07], [HJMS03] is a model checking tool for C programs and allows checking safety properties on an automatically generated abstract model of the program[2].

Java Pathfinder. Java Pathfinder [VH00], formerly based on the SPIN model checker, is now an independent model checking tool based on its own Java Virtual Machine. It can be used to search for deadlocks, uncaught exceptions (for example, due to failed assertions), or even custom properties that can be specified in a Java class[3].

2.2.2 Formal Specifications

This section introduces formal specification languages which are commonly used to express safety and liveness properties. These properties are then

[1] http://www.spinroot.com
[2] http://mtc.epfl.ch/software-tools/blast
[3] http://javapathfinder.sourceforge.net

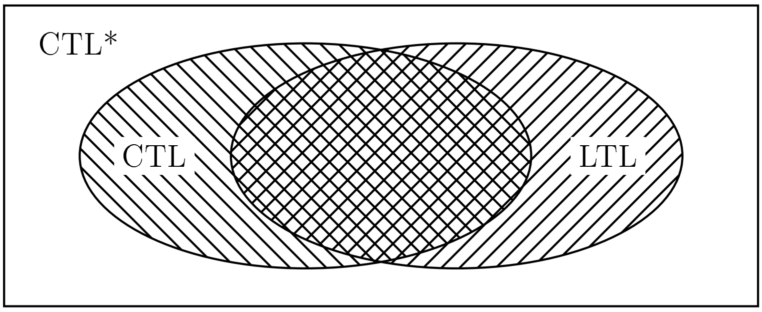


Figure 2.3: CTL* and its subsets CTL and LTL.

verified using a model checker.

Temporal Logics

Temporal logics represent propositions specifying properties of state transition systems. These properties are described in terms of sequences of transitions in the transition system using so-called *temporal operators* expressing properties like *finally* or *never*.

Computation Tree Logic* (*CTL**) is a superset of two widely used temporal logics: *branching-time logic* (CTL) and *linear-time logic* (LTL). We first describe the general properties of CTL* and then detail on the two subset languages. The relation between CTL*, CTL, and LTL is outlined in Figure 2.3.

CTL* [CGP99] formulas consist of atomic proposition symbols and the usual logic operators $\neg$, $\wedge$, $\vee$. These basic formulas are called state formulas and can be used in combination with the following temporal operators to describe properties of (infinite) paths in the computation tree.

- **X**. The subsequent formula holds at the following state (*next*).
- **F**. The subsequent formula holds at some state on the path in the computation tree (*finally*).
- **G**. The subsequent formula holds at all states on the path in the computation tree (*globally*).

- **U** binary operator: $p1\mathbf{U}p2$ means that there must exist a state at which $p2$ holds and $p1$ must hold (*until*) on all states between the current state and that state.

- **R** binary operator: $p1\,\mathbf{R}\,p2$ means that $p1$ holds up to the state where $p2$ holds (such a state does not need to exist) (*release*).

In addition, path quantifiers can be used to specify the scope of the (sub)formula. These quantifiers are **A** (all) and **E** (exists), meaning "for all computation paths" and "for some computation paths". Formulas are evaluated on a transition system starting at a specified state (usually the initial state) [CGP99].

Interface Automata

Interface automata [dAH01, dAH05] are a regular language to describe the order in which methods of a component can be called. Interface automata therefore describe in which order a component assumes that its methods are called and in which order methods of external components are called. Compatibility of two interface automata can be computed by finding an environment in which no error state is reachable (optimistic approach). The environment is defined as a sequence of external signals, e.g., a communication channel which may fail to transmit a message, whose behavior can not be guaranteed by some contract. A pessimistic approach would regard two interface automata incompatible as soon as a single environment was found in which an error state is reachable.

Contracts

Contracts introduced by Bertrand Meyer [Mey86] describe the mutual assumptions and guarantees between two components. Assumptions are expressed as preconditions, guarantees as postconditions. In addition, a contract also describes invariants that must hold at all times. Bertrand Meyer's idea is to incorporate these elements in the design process by stating the contract before coding the implementation (*design by contract*).

Design by contract is natively supported by some programming languages, like Eiffel [Mey92], D [Bri09], or Spec♯ [BLR$^+$04]. For other, more common languages, libraries and third-party tools exist, which mimic the functionality of preconditions and postconditions.

2.2.3 Satisfiability

Satisfiability (*SAT*) of Boolean properties is the decision problem of finding variable assignments that make a Boolean property true. If such an assignment can be found for all variables, the property is said to be satisfiable, otherwise it is unsatisfiable. If a formula is unsatisfiable, it is called a *contradiction*, since no assignment of truth values to its variables can make the whole formula become true.

Current SAT solvers (tools for solving satisfiability problems) are mostly SMT solvers (*satisfiability modulo theories*) supplying special theories like the theory of integers, real numbers, arrays, or bit vectors. Some of the well-known solvers are Boolector [BBL08], MathSAT [BCF$^+$08], Yices [DdM06], or Z3 [dMB08].

Most SAT solvers are based on variations of the DPLL algorithm (Davis-Putnam-Logemann-Loveland) [DP60] assigning truth values to unassigned variables, propagating implications on other variables, and then either assign truth values to other variables or backtrack in case of conflicts. Additionally, heuristics can be applied to choose those variables as assignment candidates which lead to a satisfying assignment most quickly.

2.3 Belief Revision and Belief Update

The terms belief revision and belief update can be found in disciplines like philosophy, artificial intelligence, or databases. In a nutshell, belief revision and belief update are two strategies for adding conflicting information to a knowledge base. Depending on the reason for the belief change, the one or the other belief change strategy is the better choice. This section will only consider the AI view on belief change.

2.3.1 Definitions

The following definitions give basic understanding about knowledge bases and belief change operators.

Definition 2.1 *A* knowledge base *(belief base) is a finite set of formulas consisting of a finite set of atoms ($ATM = p, q, r, ...$) and the usual logic operators $\neg$, $\wedge$, $\vee$, as well as the symbols $\top$ and $\bot$ for true and false. Knowledge bases are equal to the conjunction of their elements.*

Definition 2.2 *A knowledge base K is* consistent *if it is satisfiable.*

Definition 2.3 *A* belief change *is an operation $*$ mapping a current knowledge base K and new information N, a set of formulas, to a new knowledge base $K * N$.*

A belief change adds new information to an existing knowledge base while keeping the knowledge base consistent. If new information added to the knowledge base would make the resulting knowledge base inconsistent, some of the old information needs to be removed from the knowledge base. Belief revision and belief update are two strategies differing in how contradicting knowledge is treated.

2.3.2 Belief Revision

Belief revision ($\circ$) is the type of modification used when the change of the knowledge base is due to new information about a *static world*. The change of the knowledge base is therefore due to updated information on an *unchanged* state of the world. Alchourrón, Gärdenfors, and Makinson [AGM85] proposed 8 postulates (known as the *AGM postulates*) that every adequate revision operator should satisfy. These 8 postulates have been reformulated by Katsuno and Mendelzon to the following 6 revision postulates:

(R1) $(K \circ N) \Rightarrow N$. The result of the revision contains the new information. New information has higher priority than old information.

(R2) If $K \wedge N$ is consistent, then $K \circ N = K \wedge N$. If possible, the revision uses conjunction to add new information.

(R3) If N is satisfiable then $K \circ N$ is satisfiable. Therefore, revision always establishes a consistent knowledge base, even if the original knowledge base was inconsistent, unless N is inconsistent by itself.

(R4) If $(K_1 \Leftrightarrow K_2) \wedge (N_1 \Leftrightarrow N_2)$ then $(K_1 \circ N_1) \Leftrightarrow (K_2 \circ N_2)$. The revision operator should be invariant to the syntactic form of the new information, thus logically equivalent information results in the same new knowledge base.

(R5) $(K \circ N_1) \wedge N_2 \Rightarrow K \circ (N_1 \wedge N_2)$. A revision by $N_1 \wedge N_2$ is weaker than just adding N_2 to the knowledge base updated by N_1.

(R6) If $(K \circ N_1) \wedge N_2$ is satisfiable then $K \circ (N_1 \wedge N_2) \Rightarrow (K \circ N_1) \wedge N_2$.

(R5) and (R6) describe the rule, that the revision operator should be applied with minimal change [KM89].

2.3.3 Belief Update

Belief update ($\diamond$) is the type of modification used when the change of the knowledge base is due to new information based on changes in an *evolving world*. The change of the knowledge base is therefore due to updated information on a world that has changed since the knowledge base was established. Similar to the AGM postulates, Katsuno and Mendelzon defined 8 postulates for update operators (*KM postulates*) [KM91].

(U1) $(K \diamond N) \Rightarrow N$. The result of the update contains the new information. New information has higher priority than old information (as R1).

(U2) If $K \Rightarrow N$ then $(K \diamond N) \Leftrightarrow K$. Nothing needs to be changed, if the new information is already present in the knowledge base.

(U3) If N is satisfiable and K is satisfiable then $K \diamond N$ is also satisfiable. Therefore, update only has to establish a consistent knowledge base, if the original knowledge base and the new information were consistent.

(U4) If $K_1 \Leftrightarrow K_2 \wedge N_1 \Leftrightarrow N_2$ then $K_1 \diamond N_1 \Leftrightarrow K_2 \diamond N_2$. The update operator should be invariant to the syntactic form of the new information, thus logically equivalent information results in the same new knowledge base.

(U5) $(K \diamond N_1) \wedge N_2 \Rightarrow K \diamond (N_1 \wedge N_2)$. An update by $N_1 \wedge N_2$ is weaker than just adding N_2 to the updated by N_1.

(U6) If $K \diamond N_1 \Rightarrow N_2$ and $K \diamond N_2 \Rightarrow N_1$ then $K \diamond N_1 \Leftrightarrow K \diamond N_2$. If N_1 and N_2 are equivalent under K, then they result in the same update.

(U7) If K is complete then $((K \diamond N_1) \wedge (K \diamond N_2)) \Rightarrow K \diamond (N_1 \vee N_2)$. A knowledge base is complete, if it has a truth value for every symbol. This postulate is almost meaningless since knowledge bases are in general incomplete [HR99].

(U8) $(K_1 \vee K_2) \diamond N \Leftrightarrow (K_1 \diamond N) \vee (K_2 \diamond N)$. Updating the two alternative knowledge bases is equivalent to updating their disjunction. This postulate describes the idea of modelwise updating.

Different proposals for concrete update operations have been made. Most of the proposed operators do not fulfill all of the postulates [HR99]. Only few operators satisfy all 8 KM postulates. Therefore these postulates are discussed controversially and Herzig and Rifi [HR99] have another set of postulates deducted from the 8 KM postulates including integrity constraints [Win90, HR99] (formulas that must be guaranteed to hold after every update).

2.3.4 Winslett's Standard Semantics

Winslett's standard semantics [Win90] defines an update operator fulfilling only some of the KM postulates for update operators: (U1), (U3), (U7), and (U8). Postulate (U2) is not satisfied, because the knowledge base may be altered, even if $K \Rightarrow N$. We denote the update operator defined by Winslett as $\diamond_{WSS}$. In a nutshell, the operator replaces existing information on a symbol with new information about the symbol, and adds information about symbols not stated so far. Consider $p \diamond_{WSS} (p \vee q) = p \vee q$. This operation obviously does not satisfy (U2), since $p \Rightarrow (p \vee q)$ but $(p \vee q) \Rightarrow p$ does not hold.

Similarly, a counterexample for (U4) can be found: consider a knowledge base p and updates $q \wedge (p \vee \neg p)$ and q. The update results in $q \wedge (p \vee \neg p)$ and $p \wedge q$. Obviously, the results are not equal. This shortcoming can be easily overcome by eliminating redundant atoms.

2.3.5 Example

The following example is taken from [KM91].

Consider a room with two objects in it, a book and a magazine. Suppose b means the book is on the floor, and m means the magazine is on the floor. Then, $K = \{b \dot{\vee} m\}$ states that the book or the magazine is on the floor, but not both ($\dot{\vee}$ stands for xor). Now we order a robot to put the book on the floor. The result of this action should be represented by the update of K with $N = \{b\}$.

If we apply *revision*, the result of $K \circ N$ is $K \wedge N$, that is $(b \dot{\vee} m) \wedge b = b \wedge \neg m$. But why should we conclude that the magazine is not on the floor? If we apply *update*, the result of $K \diamond N$ is b, that is we do not know anything about m any more, which is exactly what we would expect. The difference of the two operators is therefore, that revision assumes that the new information is additional knowledge about an unchanged world, while update assumes that the new information is due to a change of the real world.

2.3.6 The Frame Problem

The *frame problem* deals with the uncertainty involved in changing parts of a world without explicitly stating which parts of the world do not change. There are different solutions to the problem from which we will only describe the one used in our implementation of the belief update.

The *default logic solution* solves the frame problem by assuming that a property not stated in the change action did not change. Thus, exactly the stated properties change and all other properties (not conflicting with the changed properties) remain unchanged.

2.3.7 Open vs. Closed World Assumption

Similar to the assumption about unstated changes to properties, we also need assumptions about how to handle properties that are not known to be true or false. Assume that we have a knowledge base consisting of the information $a \wedge b$. If we want to deduce $b \wedge c$ from this knowledge base, we need to decide whether to return $true$, $false$ or $unknown$.

Closed World Assumption

The closed world assumption presumes a complete knowledge base that contains every piece of valid knowledge. Therefore, every statement that cannot be deducted from this knowledge base must be $false$.

Open World Assumption

In contrast to the closed world assumption, the open world assumption assumes an incomplete knowledge base from which a non-inferable statement might either be due to the statement being false, or due to a missing statement. Thus, every statement that can not be deducted is said to be $unknown$ (either false or missing).

Chapter 3

MONACO

> "The most important decision in language design concerns what is to be left out."
> - *Niklaus Wirth*

The context of this thesis is the domain-specific language MONACO, a language for programming automation machines. First, the design goals of MONACO are outlined (Section 3.1). Section 3.2 and 3.3 introduce the language constructs, while Section 3.4 presents the runtime semantics of MONACO. Section 3.5 concludes with an example application. More details of MONACO are given in [PHS+08b].

MONACO (MOdeling Notation for Automation COntrol) is a domain-specific language (DSL) for programming event-based, reactive automation solutions. The main purpose of the language is to bring automation programming closer to domain experts and end users. Important design goals therefore have been to keep the language simple and to allow writing programs which are close to the perception of domain experts. The language MONACO is similar to StateCharts [Har87] in its expressive power, however, adopts an imperative notation. Moreover, MONACO adopts a state-of-the-art component approach with interfaces and polymorphic implementations and enforces strict hierarchical component architectures to support the hierarchical abstraction of control tasks. After discussing design goals, the language elements of MONACO are presented.

3.1 Design Goals

The language MONACO is designed with the goal that not only software engineers but also domain experts and, in a limited way, end users are capable of reading, writing, understanding, and adapting control programs. MONACO is specialized to a rather narrow sub-area of the automation domain, i.e., programming control sequence operations for manufacturing machines. The lower level continuous control layers and higher manufacturing execution system (MES) layers are therefore out of scope. It is intended to cover the event-based, reactive control part of machine automation software only. Therefore, a continuous control system, typically realized in languages of the IEC 61131-3 [IEC03] standard or plain C, will form a lower layer which will be controlled, scheduled, and coordinated by the higher reactive layer implemented in MONACO.

The language MONACO has been designed based on a domain analysis which showed how domain experts and end users perceive automation solutions:

- A domain expert perceives a machine as being assembled from a set of independent components working together in a coordinated fashion.

- Each component normally undergoes a determined sequence of control operations. There are usually very few sequences which are considered to be the normal mode of operation, and those are usually quite simple. Complexity is introduced by the fact that those normal control cycles can be interrupted anytime by the occurrence of abnormal events, errors, and malfunctions.

- The control sequences of the various machine components are coordinated at a higher level.

Additionally, we have identified the following requirements for a DSL and tools in the target domain:

- The language should be simple. It should contain a minimal set of language constructs and those should be intuitive and easy to understand.

- Domain experts and also end users usually have some programming experience in languages like Pascal or Basic. A syntax that is similar to one of those languages is therefore preferred.

- Reliability is more important than flexibility and expressiveness. Programs written by domain experts and end users are usually quite simple. Furthermore, end users change and adapt existing programs in a rather restricted way. However, the effect of programming mistakes can be severe.

- Reactive behavior is intrinsically complex. Especially, realizing asynchronous event and exception handling in a concise way represents a challenge.

- Programs must be runtime efficient and must usually satisfy real-time constraints.

The design of MONACO is based on the following ideas:

- Although the behavioral model of the language is very close to StateCharts, an imperative style of programming is used. The language adopts proven concepts from imperative languages such as procedural abstraction, synchronous procedure calls, parameters, block structure, lexical scoping, and a Pascal-like syntax.

- The main focus of the language is on event handling. Statements have been introduced to express reaction to asynchronous events, parallelism and synchronization, exception handling and timeouts in a concise way. However, asynchronous event handling is clearly separated from normal operation sequences to avoid mingling the normal code with exception handling code.

- Monaco pursues a component-based approach with strict modularization which allows a direct mapping of the machine structure to the software structure.

- In contrast to many other component-based approaches in this domain, MONACO pursues strict hierarchical control architectures of subordinate and superordinate components. A component relies only on the operations, state properties, and events from its subordinate components. It composes and coordinates the behavior of its subordinates and provides abstract and simplified views to its superordinate component. Thus, complex components can be built by composing existing components instead of directly controlling signals of a machine.

- The assembly of MONACO components to MONACO programs is done in a separate configuration phase (setup) prior to execution. That means the entire system is statically configured, i.e., all components, component parameters and the component hierarchy are fixed and can not change while the program is running. This static nature of MONACO programs is an important property which makes, for example, code optimization or static program analysis feasible.

In the following, the main language elements are presented.

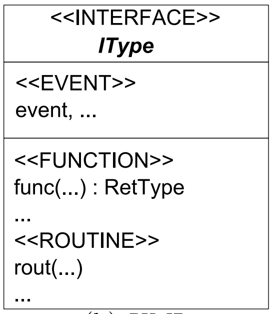
3.2 Component Approach

3.2.1 Interface Declarations

Interface declarations (Figure 3.1) are used for defining the static contract between components and their clients and hence have a similar purpose as interfaces in modern object-oriented languages. However, interfaces in MONACO account for the hierarchical communication architecture of control programs. On the one hand, an interface defines the externally visible operations of a component in the form of routine declarations. Those represent the operations a superordinate will be able to call. On the other hand, an interface defines how a component will provide feedback about the fulfillment of its control tasks. This is done by specifying events it will signal and functions it provides for accessing runtime state (properties) of the component. In other words, the routines define tasks a component can perform and the events and functions define feedback the component will provide.

```
INTERFACE IType
 EVENTS event, …;
 FUNCTION func(…):RetType;
 …
 ROUTINE rout(…);
 …
END IType
```

(a) Monaco

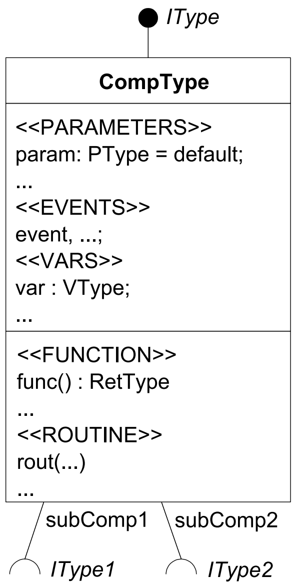


(b) UML

Figure 3.1: Interface declaration in MONACO (left) and UML (right).

```
COMPONENT CompType IMPLEMENTS IType
  PARAMETERS
    param : PType := default;
    ...
  SUBCOMPONENTS
    subComp1 : IType1;
    subComp2 : IType2;
    ...
  EVENTS
    event, ...;
  VARS
    var : VType;
    ...
  FUNCTION func(...):RetType
  BEGIN ... END
  ...
  ROUTINE rout(...)
  BEGIN ... END
  ...
END CompType
```

(a) MONACO



(b) UML

Figure 3.2: Component declaration in MONACO (left) and UML (right).

3.2.2 Component Implementations

Interfaces are implemented by components (Figure 3.2), i.e., components have to implement the routines, functions, and events defined in the interfaces. A component has parameters and internal state variables. A parameter is a runtime constant used to configure a component instance at setup time. A variable, however, is used to hold runtime state properties of a component.

Components usually rely on subcomponents to fulfill their control tasks. A component therefore declares subcomponent variables which can hold references to subcomponent instances. Interface types are used in the subcomponent variable declarations. The subcomponent declaration represents the required interfaces of the component (Figure 3.2). Subcomponents are polymorphic, i.e. any component implementing (providing) the required interface can be used. The actual subcomponent instance is plugged into the subcomponent slot at setup time (see below).

There are no access modifiers in MONACO. Only elements defined in the implemented interfaces of the component are externally visible.

Components implement functions, events and routines. A function implementation in a component is similar to functions in procedural programming languages, e.g., Pascal. They return runtime state properties of components. In MONACO, functions have no side effects, i.e., they are not allowed to set global variables, call routines, raise events, or to recurse. Usually functions are used to compute important state properties and forward those in a more abstract, concentrated form to the superior component.

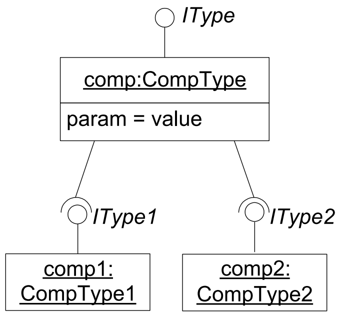
Routines are used to implement control algorithms and therefore constitute the central programming elements of components. Routines will be discussed in detail in Section 3.3.

3.2.3 Static Configuration

In order to create a complete MONACO program, MONACO components have to be instantiated and the component/subcomponent relation needs to be established (Figure 3.3). Furthermore, component parameters have to be set if the desired values differ from the defined default values. This static configuration of the system is established in a setup phase prior to program execution. The configuration cannot be changed during the execution of the MONACO program.

```
SETUP
VARS
  comp : CompType;
  comp1: CompType1;
  comp2: CompType2;
  ...
BEGIN
  comp.param := value;
  ...
  comp.subComp1 := comp1;
  comp.subComp2 := comp2;
  ...
END SETUP
```

(a) Monaco



(b) UML

Figure 3.3: Subcomponent relation in Monaco (left) and UML (right).

3.3 Reactive System Programming

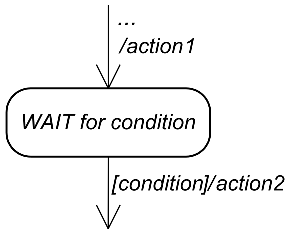
3.3.1 Control Routines

Routines are used to implement control algorithms of components. Routines are defined similar to procedures in imperative languages. They can have parameters, local variables and a body with a statement sequence. Well-known language constructs from structured programming languages like block structure, lexical scoping, loops, if statements etc. are used. Additionally, special programming constructs for parallel tasks and event handling with semantics similar to StateCharts are provided. Neither direct recursion, nor mutual recursion of routines is allowed.

Routines can be declared **ATOMIC** which means that their execution cannot be interrupted by event handlers and that they are executed atomically when used in a parallel branch. In fact, these routines may not make use of any reactive statements (such as conditional waits, parallel execution, or event handlers), but may, for example, only set a variable or call another atomic routine. Non-atomic routines may use the reactive statements as presented in Sections 3.3.3-3.3.6.

```
...
action1;
WAIT condition;
action2;
```

(a) Monaco



(b) StateCharts

Figure 3.4: **WAIT** statement in Monaco (a) and StateCharts (b).

3.3.2 Imperative Statements

Monaco comes with imperative statements like **IF** and **WHILE** used within routines to affect the control flow. Their semantics is in accordance with common programming languages.

The **IF** statement is used to conditionally execute a code block. The condition can be any Boolean expression. If the condition is not true, the **ELSE** branch of the **IF** statement is executed.

Similarly the **WHILE** statement can be used to declare a conditional repetition of a code block. The head of the **WHILE** statement contains a condition. As long as this condition is true, the block of the statement is executed.

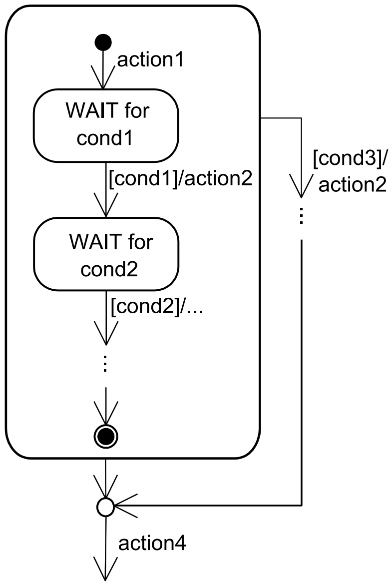
3.3.3 Conditional WAIT

The **WAIT** statement suspends the execution of the current execution thread until a specified condition is satisfied. Any Boolean expression as well as events can be used as a condition. Thus, `x>0`, `evtClosed.FIRED`, and **`TIMEOUT`**`(1000)` are all valid conditions. The latter expression returns true, as soon as the specified time in milliseconds has passed since the statement was reached.

Compared to StateCharts, a **WAIT** corresponds to a state node with the condition as the triggering event (Figure 3.4).

```
BEGIN
  action1;
  WAIT cond1;
  action2;
  WAIT cond2;
  ...
ON cond3
  action3;
  ...
END
action4;
```

(a) Monaco



(b) StateCharts

Figure 3.5: **ON** handler in Monaco (a) and StateCharts (b).

3.3.4 Asynchronous Event Handling

ON handlers are used to handle events which can occur asynchronously to normal, sequential program execution. They are similar to exceptions in general-purpose programming languages. **ON** handlers specify a condition (see valid conditions in a **WAIT** statement above) and are attached to **BEGIN**/**END** blocks (Figure 3.5). Their meaning is that, whenever the condition of the **ON** handler becomes true while program execution is within the **BEGIN**/**END** block or within a routine called in this block, the block is left and the statement sequence of the **ON** handler is executed. For **ON** handlers to be meaningful, the guarded **BEGIN**/**END** block has to have blocking statements, i.e., **WAIT** statements, where program execution gets suspended and the asynchronous event handling can occur.

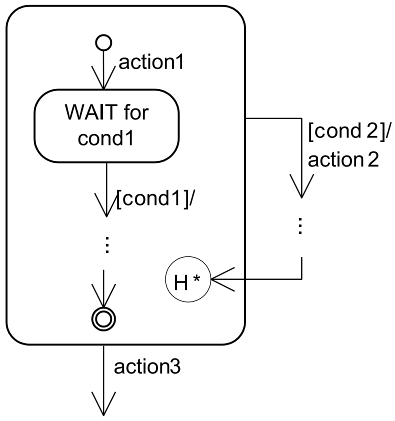
If **ON** handlers are nested, the dynamically innermost **ON** handler has precedence over outer **ON** handlers. **ON** handlers have interruptive behavior, therefore program execution continues immediately after the handler.

ON handlers show similarities to try/catch constructs in Java, however, they are much more general. While in Java an exception must be thrown

```
BEGIN
  action1;
  WAIT cond1;
  ...

ON cond2
  action2;
  ...
  RESUME;
END
action3;
```

(a) Monaco



(b) StateCharts

Figure 3.6: **RESUME** statement in Monaco (a) and StateCharts (b).

explicitly and then can be caught in catch clauses, **ON** handlers are triggered by arbitrary Boolean conditions becoming true.

ON handlers in Monaco are analogous to OR states and their transitions in StateCharts. Figure 3.5 shows the relationship. The OR state groups the states, e.g., the blocking **WAIT** statements, and transitions within the **BEGIN**/**END** block. The transition leaving the OR state is labeled with the condition of the **ON** handler. An **ON** handler can consist of an arbitrary sequence of statements.

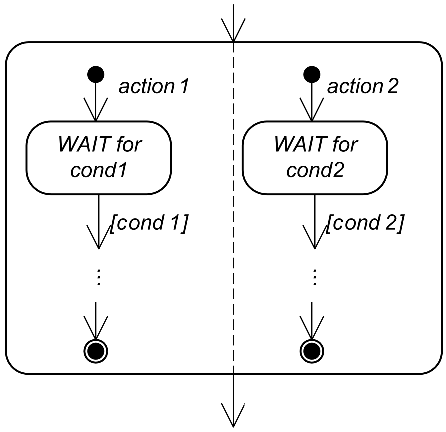
The interruptive behavior of an **ON** handler is the default. However, the **RESUME** statement can be used to resume execution of the block after the handler code has been executed. The execution of the block is resumed exactly where it was interrupted, even if it was interrupted within a routine call. The **RESUME** statement therefore has the same semantics as the deep history node in StateCharts (Figure 3.6). Currently, there is no statement equivalent to the normal history node in Monaco.

3.3.5 Parallel Execution Threads

The **PARALLEL** statement is used for creating multiple concurrent execution threads. Each parallel execution thread consists of a statement or a statement block. As soon as all parallel execution threads have terminated, program ex-

```
PARALLEL
  action1;
  WAIT cond1;
  ...
||
  action2;
  WAIT cond2;
  ...
END
```

(a) MONACO



(b) StateCharts

Figure 3.7: **PARALLEL** statement in MONACO (a) and StateCharts (b).

ecution continues after the **PARALLEL** statement. The **PARALLEL** statement has the semantics of the AND state in StateCharts, see Figure 3.7.

3.3.6 Event Signals

Although MONACO allows using arbitrary Boolean conditions as event triggers, event signals are provided. Those are similar to the event triggers in StateCharts or the signal concept in Esterel [BC85].

An event is declared as event variable in interfaces and components with the **EVENTS** keyword (see interfaces and components above). In routine bodies events can be fired using the **FIRE** statement. The event variable can then be used like any other Boolean variable in **WAIT** and **ON** handlers (Figure 3.8). In contrast to normal Boolean variables, a fired event is true for one logical time step and reset automatically in the next time step. See next section for execution details.

3.4 Execution Semantics

MONACO's execution semantics is based on the following concepts: synchronous routine calls, cooperative multitasking, fair thread scheduling, and

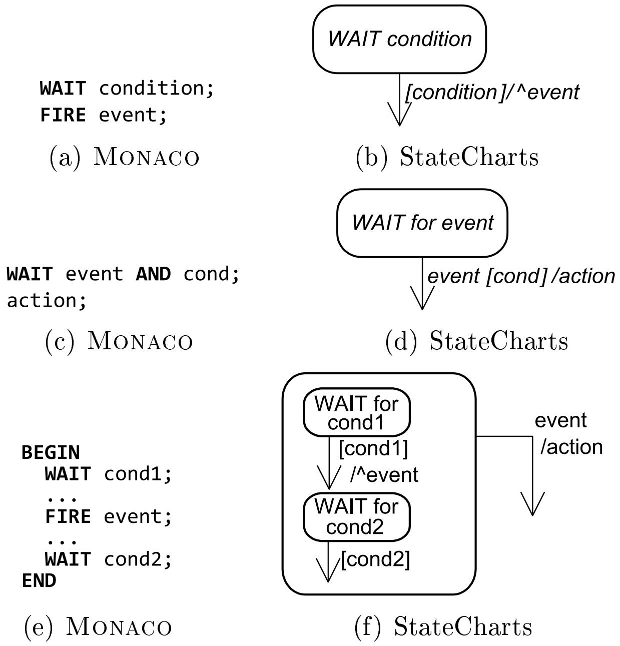


Figure 3.8: Usage of event signals with equivalent StateChart models.

event broadcast. In the following we will discuss those issues in more detail.

3.4.1 Synchronous Routine Calls

Routines are called synchronously, i.e., the caller waits until the routine terminates. This is an important difference to many component approaches in the real-time domain, e.g., UML/RT, where interaction between components happens by event signals only. We have experienced, that synchronous call semantics together with the hierarchical communication architecture lead to control programs which are easier to comprehend by domain experts and end users (see example in Section 3.5).

3.4.2 Cooperative Multitasking With Fairness

Monaco employs a cooperative multitasking scheme with fairness. There are well-defined scheduling points in a program where threads can get suspended and other threads get the chance to proceed. Scheduling points are **WAIT**

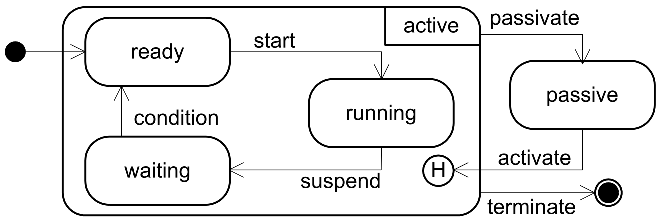


Figure 3.9: Thread state diagram (simplified).

statements, points before and after a **PARALLEL** statement, and at routine returns. Between those points program execution is treated as atomic and cannot be interrupted. Therefore, program execution is analogous to the *run-to-completion* semantics of StateCharts [Har87].

Threads are created in MONACO by the **PARALLEL** statement and **ON** handlers. A **PARALLEL** statement creates a thread for each branch which is ready for execution. The main branch is then suspended until all branches are terminated. Similarly, an **ON** handler creates a thread which is waiting for its condition. **ON** handler threads are terminated when execution of the guarded block has finished, regardless of whether the handler thread was executed.

A fixed precedence order is used to arbitrate between competing parallel threads. Currently, the order is determined based on order in which the parallel branches appear in the source code. Furthermore, **ON** handlers always have precedence over their main thread and, in case of nested active handlers, the innermost handler in terms of the dynamic nesting is preferred. This approach is simple and deterministic and we have experienced that it serves our objectives. For more details on the execution semantics refer to Section 5.2 and [PHS$^+$08b].

Figure 3.9 shows state transitions of threads. Initially, each thread is in the *ready* state. This means it is not waiting for any condition and is therefore ready to run. When the scheduler starts a thread, it transits to the *running* state. It remains running until it reaches a scheduling point; it changes into the *waiting* state again. The thread becomes ready again, as soon as its condition (from **WAIT** statement or **ON** handler) becomes true.

When a thread reaches a parallel statement, it is passivated. This means it can not run until it is activated again. The thread is activated again when all branch threads are terminated.

The cooperative scheduler uses a fair thread scheduling algorithm based on logical time steps. Once started by a fulfilled **WAIT** condition, a thread only runs to the next scheduling point. At this point another thread in the *ready* state gets the chance to run. When all threads in the *ready* state have run to their next scheduling point, the logical time step is over. Therefore, when a thread is *running* once in a logical time step, it can not get started again in the same logical time step. This mechanism prevents starvation of parallel threads. It ensures that each parallel thread that is ready has a chance to run before another thread is started a second time.

3.4.3 Event Broadcast

Events are broadcast within their dynamic scope. The dynamic scope of the event is the component in which the event is declared, as well as in components using this component (only if the event is also declared in the component's interface).

Events are active for one logical time step only. That means when several **WAIT** statements and **ON** handlers are concurrently waiting for an event, they get started based on the scheduling scheme as outlined above. Moreover, events are always propagated from the innermost block outward. When an inner **ON** handler handles the event, further surrounding **ON** handlers will not receive it. Note, that this behavior only applies to events since events are deactivated once they are handled. If, however, two nested **ON** handlers both wait for a Boolean condition, the outer handler may be activated after the inner handler was activated, if the condition is still true.

3.5 Example Control Program

This section demonstrates programming in Monaco with a sample application. It shows how language constructs presented in this chapter are employed in realizing a component-based, hierarchical control program. First, we briefly describe the physical process of injection molding. Next, we show the decomposition of the machine into a hierarchy of components, and then show the hierarchical abstraction of control functionality by components at different hierarchy levels.

3.5.1 Example System

For validation of concepts, we have developed several example applications in MONACO. One has been a reimplementation of an existing control program for an injection molding machine, which was originally implemented in the IEC 61131-3 [IEC03] standard languages. We have implemented the event-based part of the application in MONACO and have coupled it with a simulator for testing purposes. The MONACO program has led to a drastic reduction in code size to less than one fifth of the original code, and, at the same time, to a significant improvement in code clarity. Special emphasis has been put on handling errors and malfunctions of the machine. It has been shown that the MONACO language is capable of describing machine failure handling in a compact and concise way. In the following we show code fragments of a simplified version of the example software system.

Our example deals with injection molding machines. These machines are used to produce plastic parts by injecting heated, semi-fluid plastic into a mold where the plastic cools down and hardens within a short period. In order to produce plastic parts with various notches and holes, it is necessary to have an adaptable mold that inserts so-called cores into the molding chamber during the injection process. After the plastic part is hardened, the cores are removed, the part gets ejected, and the process starts over again. During the cooling phase, new raw material (plastic pellets) is heated up for the next injection phase.

Figure 3.10 shows the structure of the sample molding machine. There are two main components in the machine: the mold subsystem with the clamp, the ejector and a core puller; and the nozzle subsystem that is mounted on a sledge with the material funnel, the heating system and the screw for injection. Finally, the ejector serves the purpose of ejecting the finished parts out of the mold.

3.5.2 Component Hierarchy

The component hierarchy of the control program resembles the structure of the real machine (Figure 3.11). There is a direct mapping from the problem structure to the solution structure. On top, the `Machine` component is responsible for encoding the overall control cycles. It knows different operation

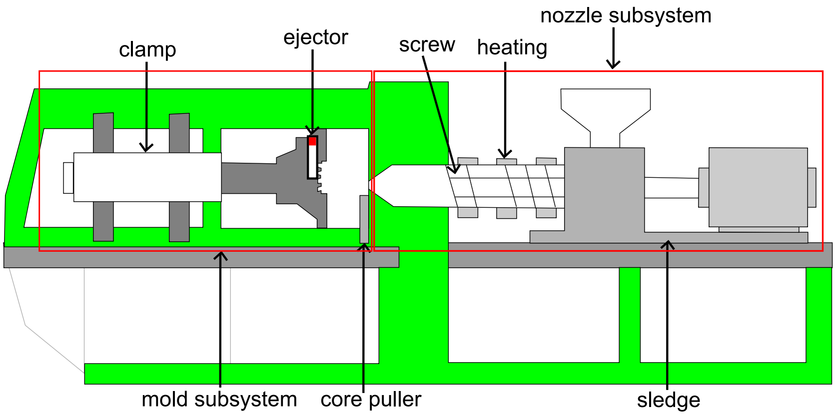


Figure 3.10: Structure of the molding machine.

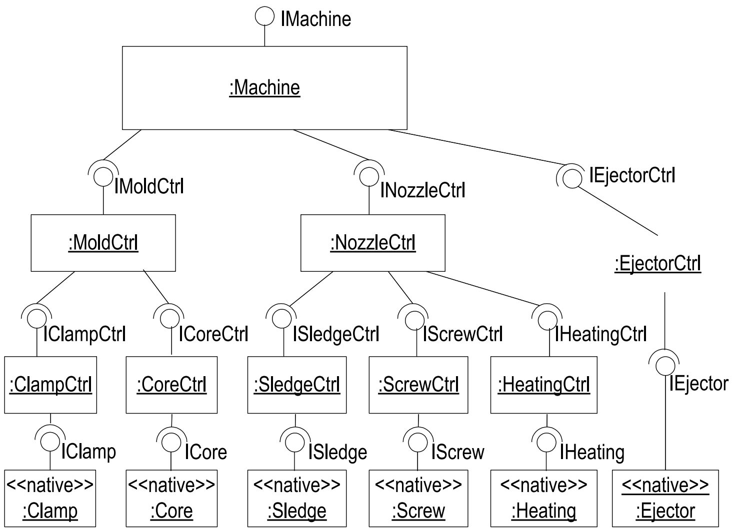


Figure 3.11: Component hierarchy of the molding machine.

modes, e.g., full automatic or half automatic and relies on and coordinates several subcomponents corresponding to the different machine subsystems. The components for nozzle and mold are further decomposed according to the different parts of the subsystems. At the bottom of the hierarchy there are components for interfacing with lower level control layers or the hardware. Those are usually implemented in the native language of the lower layers; in this example program Java components build the interface to the simulator.

Components at different hierarchy levels typically serve different purposes as follows:

- Components at the bottom are used for interfacing with the hardware or lower control layers. They usually set and read basic system variables. This layer is often referred to as hardware integration layer.
- Components at the first level compose primitive operations of the bottom layer into elementary control routines and supervise their execution.
- Higher up in the hierarchy there are several coordination components which coordinate and supervise the operations of several subcomponents.

3.5.3 Control Components

Interface to hardware and continuous control layers

In the example program, the components forming the leaves of the component hierarchy are native Java classes building the interface to a simulator which simulates the real machine and the continuous control layer. Native components implement a Monaco interface which represents the interface for the components higher in the component hierarchy (there is direct mapping of routines, functions and events to equally named Java methods). The following code snippet (Figure 3.12) shows the interface definition of the core puller component `ICore`. The interface defines elementary routines to set system variables to start and stop insertion and removal of the core and a function giving the current position of the core puller.

First level control components

The components residing in the hierarchy level directly above the native components use those interfaces to compose elementary operations into basic task routines. For example, the `CoreCtrl` component has the native component `core` as its single subcomponent. It defines two routines to insert and remove the core. Additionally, a stop routine is provided which immediately stops all movements.

```
INTERFACE ICore
  FUNCTION position() : REAL;

  ROUTINE startInsert();
  ROUTINE stopInsert();
  ROUTINE startRemove();
  ROUTINE stopRemove();
END
```

Figure 3.12: Interface `ICore`.

```
COMPONENT CoreCtrl IMPLEMENTS ICoreCtrl
  PARAMETERS
    coreMovementStartedTimeout : INT := 200;
    coreInsertTimeout : INT := 1400;
    coreInsertedPos : REAL := 0.6;
    coreRemovedPos : REAL := 0.8;
  SUBCOMPONENTS
    core : ICore;
  EVENTS error;

  FUNCTION isInserted() : BOOL
  BEGIN
    RETURN core.position() >= coreInsertPos;
  END inserted
  ...
END CoreCtrl
```

Figure 3.13: Component `CoreCtrl`.

The following code snippet (Figure 3.13) shows part of the `CoreCtrl` component. Besides showing declaration of parameters, subcomponents and events, it also demonstrates how functions are employed for abstracting state properties from lower level information of subcomponents.

Routines implement the basic control tasks. However, besides defining the basic sequence of actions, routines also check for the correct execution of control tasks and correct reactions from the subordinate. This can be done using **ON** handlers.

The code snippet (Figure 3.14) demonstrates this approach with the

```
ROUTINE insert()
BEGIN
  core.startInsert();
  BEGIN
    WAIT NOT core.isRemoved();
  ON TIMEOUT(coreMovementStartedTimeout)
    stop();
    FIRE error;
    RETURN;
  END
  BEGIN
    WAIT core.isInserted();
    core.stopInsert();
  ON core.isRemoved()
    stop();
    FIRE error;
    RETURN;
  ON TIMEOUT(coreInsertTimeout)
    stop();
    FIRE error;
    RETURN;
  END
END insert
```

Figure 3.14: Routine `insert`.

`insert` routine. First, `startInsert` is called for the subcomponent `core` which will set a hardware signal and start the insertion process. Next, a reaction from the `isRemoved` signal is expected. If this sensor does not go to false within a given (short) time period, a fault in the insertion process or a faulty sensor has to be assumed; so the process is stopped and an error event is fired. Next, the `insert` routine waits for the `isInserted` signal to become true and then stops the insertion process. Again the process is supervised by two **ON** handlers. The first handler checks that the `isRemoved` signal does not switch to true again (which might result from a faulty sensor). The second handler checks that the reaction of the `isInserted` signal occurs in time. In both error cases the process is stopped and the `error` event is fired. Note, that this way, the `insert` routine is guaranteed to either run correctly to its end or an error signal will occur.

The control behavior defined so far is provided in a more abstract way in an interface declaration to the upper component. The following code snippet (Figure 3.15) shows the interface `ICoreCtrl` of the `CoreCtrl` component. There are routines for inserting, removing, and stopping the core, as well as two Boolean functions telling if the core is inserted or removed. Additionally, the `error` event appears in the interface which means that the upper component will be able to check for the errors occurring during execution of the control routines.

```
INTERFACE ICoreCtrl
  EVENTS error;
  FUNCTION isInserted() : BOOL;
  FUNCTION isRemoved() : BOOL;
  ROUTINE insert();
  ROUTINE remove();
  ROUTINE stop();
END ICoreCtrl
```

Figure 3.15: Interface `ICoreCtrl`.

Coordination levels

As next higher level component the `MoldCtrl` component is discussed. This component has to coordinate the operations of the `core` and the `clamp` subcomponents (see Figure 3.16).

The code snippet in Figure 3.17 exemplifies this by the `close` routine. Its purpose is to control the process of closing the clamp and inserting the

```
COMPONENT MoldCtrl IMPLEMENTS IMoldCtrl
  PARAMETERS
    coreInsertPos: REAL := 150;
  SUBCOMPONENTS
    clamp : IClampCtrl;
    core : ICoreCtrl;
  ...
END MoldCtrl
```

Figure 3.16: Component `MoldCtrl`.

```
ROUTINE close()
BEGIN
  PARALLEL
    clamp.close();
  ||
    WAIT clamp.position() >= coreInsertPos;
    core.insert();
  END
ON core.error OR clamp.error
  stop();
  FIRE error;
  RETURN;
END close
```

Figure 3.17: Routine `close`.

```
INTERFACE IMoldCtrl
  EVENTS error;
  FUNCTION isOpen() : BOOL;
  FUNCTION isClosed() : BOOL;
  FUNCTION clampPos() : REAL;
  ROUTINE open();
  ROUTINE close();
  ROUTINE stop();
END IMoldCtrl
```

Figure 3.18: Interface `IMoldCtrl`.

core, which should occur in parallel. However, insertion of the core has to start after the clamp has reached the `coreInsertPos`. In this routine we do not need to worry about timeouts and possible error conditions of the core or any other subcomponent. Those routines are already checked for correct execution and fire error events. Thus, it is sufficient to have an **`ON`** handler for errors reported by the `core` and `clamp` subcomponents (which in this example again fires an event to inform its upper component). In this way, one gets a more abstract view of a subsystem. The code in Figure 3.18 shows the interface of the `MoldCtrl` component.

Finally, the following routine `automatic` represents the overall automatic control cycle of the machine (Figure 3.19). This is usually the level which is also presented to end users. The operation cycle of the machine gets clearly represented in the code. In the inner control loop first the mold is closed. Then injection is done and in parallel the cooling time is checked. Then, in parallel activities, the mold is opened, new material is inserted into the screw (`nozzle.plasticize`) and, after the mold has been opened to a determined point, the piece is ejected.

```
ROUTINE automatic()
BEGIN
  BEGIN
    nozzle.startHeating();
    WAIT nozzle.temperatureReached(nomTemp);
    LOOP
    BEGIN
      mold.close();
      PARALLEL
        nozzle.inject();
      ||
        WAIT TIMEOUT(coolingTime);
      END

      PARALLEL
        nozzle.plasticize();
      ||
        mold.open();
      ||
        WAIT mold.clampPos() < 0.5;
        ejectorCtrl.eject();
      END
    END

  ON mold.error OR nozzle.error OR systemStopped()
    PARALLEL
      mold.stop();
    ||
      nozzle.stop();
    ||
      ejectorCtrl.stop();
    END
  END
  nozzle.stopHeating();
END automatic
```

Figure 3.19: Routine `automatic`.

Chapter 4

Contracts and Constraints

This chapter introduces contracts as a mean for specifying component behavior as well as constraints that describe dependencies between components. First, Section 4.1 discusses contracts and their relation to MONACO components. Section 4.2 introduces an LTS-based automata formalism used to specify component behavior. The presented automaton formalism is augmented with pre- and postconditions, as well as invariants in Section 4.3. Constraints (safety properties) are presented in Section 4.4. Finally, Section 4.5 briefly describes notations for contracts and constraints.

4.1 Introduction

In general, contracts are formal agreements between two or more parties. Bertrand Meyer introduced the paradigm of *Design By Contract* [Mey86] which defines contracts as specifications that describe as closely as possible the mutual obligations and benefits involved in the communication between software elements.

This definition comprises more than usual interfaces in object-oriented programming languages or MONACO. Interface definitions usually define routines and functions with their parameter types and return values. While this description states what can be done with an object of this type (structure, static behavior), it does not state anything about the effects, valid sequences (dynamic behavior), and valid state of routine calls. That is, it only specifies the syntax and says nothing about the behavior of components.

In contrast, protocol contracts as introduced in this thesis, define the dynamic behavior in the communication between software elements. They are similar to behavior protocols [PV02], sequencing constraints in Cecil [OO90], and interface automata [dAH01] (see Section 9.1). Protocol contracts therefore allow one to express the following aspects of the dynamic behavior of components:

Valid call sequences. Operations of components often require a certain sequence in order to be successful. For example, a component's behavior often consists of an initialization phase, several operative actions, and eventually a termination phase. If this sequence is not obeyed, runtime errors occur, or in the domain of industrial automation, a machine can be damaged. It is therefore desirable to explicitly state these restrictions on the component usage and to be able to check and enforce these sequences.

Effects of a call. Routine calls normally result in changes of the component state. These changes (the *effects* of the routine) are called *guarantees* or *postconditions* and can be expressed by Boolean conditions that are guaranteed to hold after the call to the routine.

Requirements of a call. In order to be executable, routines may require the component to be in a certain state. Such a requirement is called a *precondition*. A precondition is expressed as a Boolean condition that needs to hold before a call to the routine can be executed.

Initial state of a component. In order to deduce the situation of a component at a certain position in the execution, it is necessary to define the initial situation, i.e. the state of the component before any routine of the component has been called.

Invariants. Invariants in protocol contracts describe immutable propositions that help reasoning about component states by adding information about the dependencies of component properties. The dependencies can be caused by physical exclusion of states.

In MONACO we use protocol contracts as outlined above to constrain call sequences and to specify the dynamic behavior of components. In doing so, we

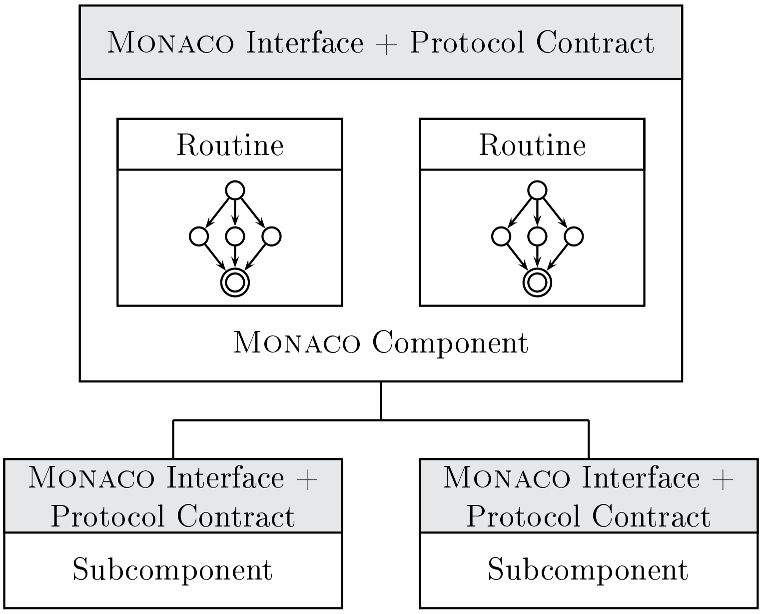


Figure 4.1: Protocol contracts in the MONACO component hierarchy.

exploit the hierarchical structure of MONACO components. Since each component implements an interface, and subcomponents are specified by their interface type, the MONACO component hierarchy encapsulates components as illustrated in Figure 4.1. The figure shows a component with two routines, and two subcomponents each specified by their interface. The interfaces of the subcomponents each have a contract describing how the subcomponents can be used. The component itself also implements an interface and a contract. The contract of the component defines how its routines can be called.

In the following, we introduce protocol contracts for MONACO components which are based on *labeled transition systems* (LTS) [BJK+05].

4.2 Automata Formalism

This section reviews the well-known automata formalism *labeled transition systems* (LTS) [BJK+05] and introduces a MONACO-specific extension of LTS which is used to capture the component behavior by encoding it as valid event sequences.

Definition 4.1 *A labeled transition system is a quadruple $L = \langle S, I, A, T \rangle$ that consists of the following elements:*

- *S is the set of states.*
- *$I \subseteq S$ is the set of initial states.*
- *A is the set of actions (labels).*
- *$T \subseteq S \times A \times S$ is the transition relation.*

In contrast to finite automata, LTS do not have final states, since they help reasoning about sequences of events, not about language acceptance. Figure 4.2 shows an example of a labeled transition system consisting of three states $S = \{1, 2, 3\}$, the initial states $I = \{1\}$, the actions $A = \{a, b, c\}$, and the transition relation $T = \{(1, a, 2), (2, b, 3), (3, c, 1)\}$.

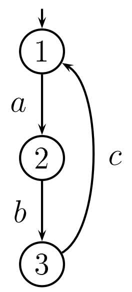


Figure 4.2: Labeled transition system.

To serve our special requirements of specifying MONACO component contracts, we extend LTS as follows. Routine calls in MONACO have synchronous semantics and can be aborted during execution. This semantics has to be reflected in the specialized LTS by separating routine calls and routine returns. The set of actions will be constrained to contain only routine calls, routine returns, events and an unobservable internal event. First, we formally introduce a MONACO component interface.

Definition 4.2 *Let $I = \langle R, F, E \rangle$ be the description of a component interface where the elements R, F, E have the following meaning:*

- *R is the set of routine symbols.*

- *F is the set of function symbols.*
- *E is the set of event symbols defined in the interface.*

Remark: We disregard parameters in the description of functions and routines. Parameters play a minor role in MONACO programs, while disregarding parameters eases the description of contracts.

Definition 4.3 *We call our extension of LTS* protocol automata. *A protocol automaton is a quadruple $PA = \langle S,\, s^{init},\, A,\, T \rangle$ describing an LTS with only a single initial state and a constrained set of actions.*

- *S is the set of states.*
- *$s^{init} \in S$ is the initial state. In contrast to LTS, we only need exactly one initial state as a component typically has exactly one initial state.*
- *$A = R \times \{call, ret\} \cup \{\tau\}$ is the set of actions (alphabet). R is the set of routine symbols defined in the interface of a* MONACO *component (see above). τ is the empty action representing an unconditional, immediate transition.*
- *$T \subseteq S \times A \times S$ is the transition relation.*

The set of actions A can be further subdivided into the sets $A_{call} = R \times \{call\}$ and $A_{ret} = R \times \{ret\}$. These sets are called the sets of call actions and return actions. Similarly, the set $T_{call} = S \times A_{call} \times S$ and $T_{ret} = S \times A_{ret} \times S$ are called the set of call and return transitions, respectively.

The separation of routine calls and routine returns is illustrated by two examples in Figure 4.3. Example (a) shows a protocol automaton consisting of three states and two transitions. State 1 is the single initial state. The two transitions represent execution of routine $r1$. The call is separated into the call and the return from the call. The first transition is a call transition, while the second is a return transition. Figure 4.3 (b) shows a protocol automaton similar to the one in (a). The difference is in the call to routine $r1$, which can either return $(r1, ret)$ or be aborted. The additional transition from state 2 to state 3 is a τ transition describing an unobservable internal event. The use of τ transitions will be explained in Section 4.3.

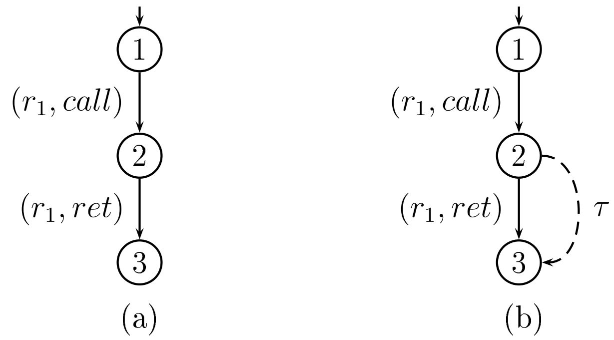


Figure 4.3: Protocol automata showing the separation of routine call and routine return and different types of transitions.

4.3 Interface Contract

We use contracts to describe valid call sequences of routines for a MONACO interface. They are based on the notion of protocol automata presented above (Section 4.2), but have additional information like preconditions and postconditions stating required states and guarantees about the behavior of a component.

4.3.1 Pre-, Post-, and Initial-Conditions

Contracts contain pre- and postconditions to express requirements and guarantees of component properties in certain states. Guarantees can be explicitly canceled using retraction, and guarantees about the initial values of component properties can be made. These conditions are reflected in a contract by the functions *Pre*, *Post*, *Retract*, and *Initial*.

Pre- and postconditions are logical propositions over all function symbols plus numerical and Boolean constants. That means we use the function symbols from F as logical variables. Functions with numerical return type can be used with relational operators and numerical constants. We allow the combination of logical expressions with the logical operators $\wedge$, $\vee$, and $\neg$.

We denote the set of all satisfiable logical propositions over symbols $f \in F$ for an interface I as C.

Definition 4.4 *Let S be the states of a protocol automaton. Then we define* *four functions:*

- $Pre : S \rightarrow C$ *is the function mapping states to the set of preconditions. The semantics of a precondition of a state is that this condition must be fulfilled before the state can be reached (i.e. the transition leading to the state can be executed).*
- $Post : S \rightarrow C$ *is the function mapping states to the set of postconditions with the meaning that the given condition is guaranteed to be true after the state is left (i.e. the transition leaving the state is executed).*
- $Retract : S \rightarrow \mathcal{P}(F)$ *maps states to function symbols. The semantics of retraction of a function symbol is, that any guarantee about this symbol is retracted.*
- $Initial \in C$ *describes the initial conditions holding before any routine has been called. This description is called* initial condition *and can be regarded as a guarantee, that the component initially is in a certain state.*

By default, a guarantee holds, until it is invalidated by a more recent guarantee. For details about knowledge update and retraction, refer to Section 6.3.

4.3.2 Invariants

Components have state properties with logical dependencies on each other. A dependency is often due to physical laws prohibiting concurrent presence of two states. These dependencies can be formulated as Boolean formulas, called *invariants*. In the literature, such invariants are also referred to as *integrity constraints* [HR99, Win90]. If these invariants are stated explicitly, they help in the knowledge deduction process by adding additional knowledge and keeping the knowledge base consistent.

For example, let's assume we have a hydraulic cylinder component that can be opened and closed. Its observable properties are the Boolean functions `isOpen` and `isClosed`. Both properties can never be true simultaneously. Yet, it is possible that the component is neither opened nor closed (it is in

```
[Invariant: NOT (isOpen() AND isClosed())]
```

Listing 4.1: Invariant describing the logical dependency between *isOpen* and *isClosed*

some intermediate position). An invariant describing the dependency of these properties together with the knowledge of one of the properties allows us to deduce that the other property does not hold. Listing 4.1 shows an example invariant describing the logical dependency between the two properties mentioned above.

Definition 4.5 *We associate a set of* invariant *conditions Inv with an interface contract. Inv* $\in C$*, that means invariant conditions are logical propositions over the function symbols* F *(see above).*

4.3.3 Summary

In summary, an interface contract consists of the following elements:

- $PA = \langle S, s^{init}, A, T\rangle$ is the protocol automaton defining valid call sequences.
- $Pre : S \rightarrow C$ is the function mapping states to the set of preconditions.
- $Post : S \rightarrow C$ is the function mapping states to the set of postconditions.
- $Retract : S \rightarrow \mathcal{P}(F)$ is the function mapping states to propositional symbols for retraction of knowledge.
- $Inv \in C$ is the set of invariant conditions (integrity constraints).

4.3.4 Examples

In the following, two example contracts will be presented, showing pre-, post- and initial conditions, as well as invariants. An example for knowledge retraction is presented in Section 6.3.2. The first example shows a contract for a hydraulic cylinder. The cylinder can be opened and closed. The interface of

```
INTERFACE ICylinder
  ATOMIC ROUTINE startOpen();
  ATOMIC ROUTINE startClose();
  ATOMIC ROUTINE stop();

  FUNCTION isOpen() : BOOL;
  FUNCTION isClosed() : BOOL;
END ICylinder
```

Listing 4.2: Interface of a cylinder component

the cylinder component is shown in Listing 4.2. The routines `startOpen`, `startClose`, and `stop` atomically start or stop a movement of the cylinder. The Boolean functions `isOpen` and `isClosed` return whether the cylinder is fully opened or fully closed.

To make the graphical representation more readable, transitions in the graphical representation of protocol automata will be labeled with $r!$ for routine calls (instead of $(r, call)$) and $r?$ for routine returns (instead of (r, ret)).

The contract for the `ICylinder` interface is as follows: the cylinder can be opened with the routine call `startOpen` and closed with a call of the routine `startClose`. The effect of a routine call is that the opening respectively closing movements are started and the routine call immediately returns. The movement can be stopped using the routine `stop`.

The two functions of the interface report whether the cylinder is currently opened, closed, or neither opened nor closed (both functions return false). Figure 4.4 shows the protocol automaton for this contract. Note that the contract does not state that the `startClose` routine causes the `isClosed` function to evaluate to true. The only conclusion that can be made is that starting the close movement makes `isOpen` evaluate to false. Note, that the postconditions are associated with the states representing the execution of the routine. The postconditions hold, as soon as this state is left.

Additionally an invariant states that the cylinder can never be open and closed simultaneously. The invariant is given by $Inv = \{\neg(isOpen \wedge isClosed)\}$.

The second example describes a contract for a driller machine like the one shown in Figure 4.5. The machine consists of two subcomponents, a driller and a cooler. The interface `IDriller` of the driller component declares

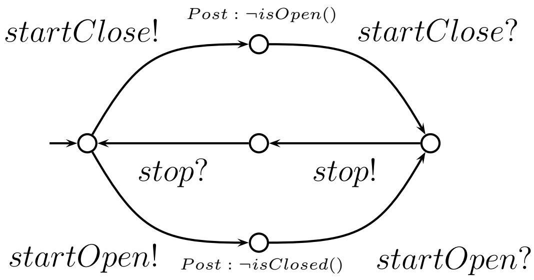


Figure 4.4: Protocol automaton for the `ICylinder` interface.

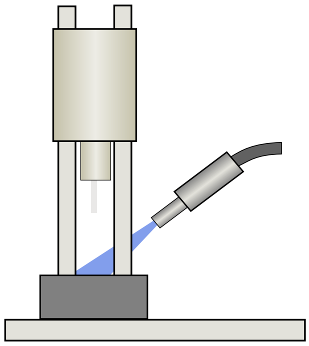


Figure 4.5: Driller and cooler component

routines and functions as outlined in Listing 4.3. The intended behavior of the interface is, that any component implementing this interface should first be started, then be moved down and up in turn and eventually be stopped. The behavior is illustrated by the protocol automaton in Figure 4.6. It contains postconditions that guarantee the effects of execution of the routines and has the initial condition $\neg isStarted()$. Moreover, the call of routine `down` has a precondition requiring that a certain revolution speed must be reached ($rpmReached()$).

The interface `ICooler` for the cooler component declares the routines `start` and `stop`, as well as the function `isCooling`. The cooler component keeps the temperature of the driller at an acceptable level. Its behavior is described by the protocol automaton shown in Figure 4.7. It describes that the cooler can be started and stopped. Additionally, the effects of the two routines are specified as postconditions.

```
INTERFACE IDriller
  ATOMIC ROUTINE start();
  ATOMIC ROUTINE stop();
  ATOMIC ROUTINE down();
  ATOMIC ROUTINE up();
  FUNCTION isStarted() : BOOL;
  FUNCTION isDrilling() : BOOL;
  FUNCTION rpmReached() : BOOL;
END IDriller

INTERFACE ICooler
  ATOMIC ROUTINE start();
  ATOMIC ROUTINE stop();
  FUNCTION isCooling() : BOOL;
END ICooler
```

Listing 4.3: Interfaces of a driller and a cooler component

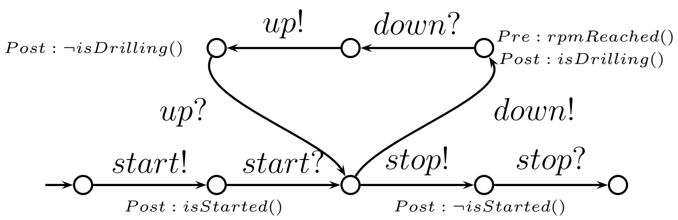


Figure 4.6: Protocol automaton for the `IDriller` interface.

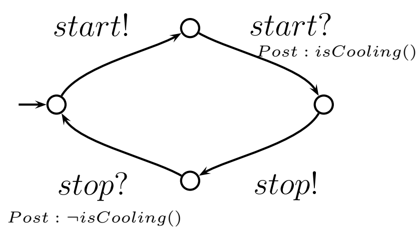


Figure 4.7: Protocol automaton for the `ICooler` interface.

4.4 Constraints

Propositional constraints describe safety properties (refer to Section 2.2.1) that must be true in every state of the system ("something bad will never happen"). In contrast to invariants, constraints are not maintained by the physical world, but rather describe that possibly fatal states must not be reachable.

Constraints define relationships between several components and therefore do not belong to a contract of a single component. For example, imagine a component having multiple subcomponents. The subcomponents are independent as they have separate contracts describing their local behavior, disregarding the existence of other components. This strict separation of components allows for simple exchange of component implementations. Nevertheless, it is necessary to provide mechanisms to synchronize two or more contracts, i.e., to describe states that the combination of those components should never reach.

Let's assume, there is a component c with subcomponents with interfaces $I_1, I_2, \ldots I_n$ where each interface I_i consists of the elements $I_i = \langle R_i,\ F_i,\ E_i \rangle$. Then we associate with the component c a constraints $Constr_c$ being a logical proposition over symbols $f \in \bigcup_i F_i$.

Assume we have a drilling machine as defined above. In this example, a constraint is that the driller must not be drilling before the cooler is cooling. Similarly, the cooler must not be stopped, while the driller is drilling. Thus, the proposition describing this constraint is $\neg(driller.isDrilling() \wedge \neg cooler.isCooling())$.

Remark: In order to avoid name clashes in constraints, function symbols are qualified with the name of the subcomponent they belong to.

4.5 Notations

In the following, we introduce two different notations for describing contracts. The first notation only allows us to describe valid call sequences. The second notation is more powerful and allows specifying all aspects of a contract.

4.5.1 EBNF Notation

This notation is based on the standard meta language EBNF (*Extended BNF*, ISO 14977 [ISO96b]). The notation does not make use of non-terminal symbols, but each production describes the complete contract for an interface as a regular expression. The terminal symbols allowed are all routine names

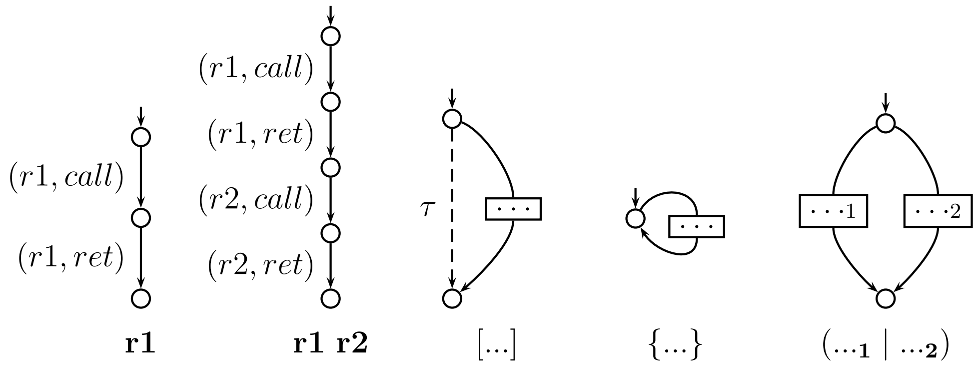


Figure 4.8: Translation of EBNF to protocol automata (... stands for an arbitrary subexpression).

in the set R (see protocol automata above), denoting the routines of the MONACO interface, to which the contract belongs.

The following EBNF metasymbols are available (... stands for an arbitrary subexpression):

- [...] The contained subexpression is optional.
- {...} The contained subexpression can be repeated arbitrary many times (including zero times).
- (...) Groups subexpressions.
- (... | ...) Separator for alternative subexpressions. The subexpressions are chosen nondeterministically.
- . (period) Terminates the definition of a protocol contract.

The conversion of terminal symbols and the metasymbols into protocol automata is straight-forward. Figure 4.8 shows the resulting protocol automata for single symbols, symbol sequences and the presented metasymbols. Routine symbols are converted into an automaton consisting of three nodes, connected by a call and a return transition. The first node is the initial state, the intermediate state represents the running routine, the last state is the state after the routine is executed. Sequences of terminal symbols are translated by creating the protocol automata of individual symbols and then merging the end state of the first symbol's protocol automaton with the initial state of the second symbol's protocol automaton. The metasymbols for

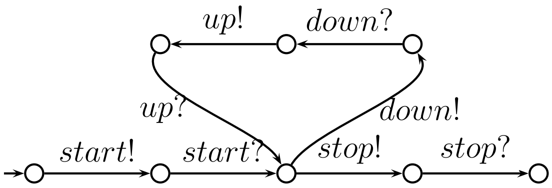


Figure 4.9: Protocol automaton resulting from Listing 4.4.

optionality add a τ transition from the initial state to the end state of the subexpression, thus allowing to omit the subexpression. The metasymbols for repetition merge the initial and the end state to a common state which is the initial state of the resulting automaton. Alternative subexpressions are created by merging all initial states and all end states of the subexpressions. Appendix C gives a full listing of the grammar of the EBNF notation.

The EBNF notation is demonstrated by the following example. Let's assume we have an interface `IDriller` declaring the routines `start`, `stop`, `down`, and `up`. The intended behavior of the interface is, that any component implementing this interface should first be started, then be moved down and up in turn and eventually be stopped. The protocol contract for `IDriller` in EBNF notation is listed in Listing 4.4.

```
IDriller = start { down up } stop .
```

Listing 4.4: Contract for `IDriller` in EBNF notation

Figure 4.9 shows the protocol automaton resulting from the contract for the `IDriller` interface.

4.5.2 Detailed Protocol Contract Notation

This notation explicitly enumerates all states of the protocol automaton, together with all transitions between the states and the initial, pre-, and postconditions as well as the invariants.

The notation starts with the declaration of the MONACO interface, followed by the initial condition, the invariants and a list of state declarations. A state declaration declares a state with a unique identifier (unique within the protocol contract) followed by a list of pre- and postconditions for the state. Then all outgoing transitions are listed. A transition is either a routine call, or a routine return, specified with the routine name followed by a `!` or

```
Interface IDriller d [Initial: NOT d.isStarted()]:
initial s0 = start!s1.
s1 [Post: d.isStarted()] = start?s2.
s2 = stop!s3 down!s4.
s3 [Post: NOT d.isStarted()] = stop?s7.
s4 = down?s5.
s5 = up!s6.
s6 = up?s2.
s7 = .
```

Listing 4.5: Contract for `IDriller` in detailed protocol contract notation

a `?` respectively, or a τ-transition.

Imagine that we want to extend the protocol contract in Figure 4.9 by adding the state property `isStarted`, modeled as a Boolean function in the `IDriller` interface. Listing 4.5 shows this extended protocol contract for `IDriller` in the detailed protocol contract notation. The resulting contract is pictured in Figure 4.10. Note that states $s1$ and $s3$ now have a postcondition.

For sake of brevity, names of states are chosen very short. For a better readability one would choose more descriptive state names like *init*, *starting*, *started*, and so forth.

In summary, the detailed protocol contract notation is much more expressive, since it can be used to describe all features of a contract. In practice, one would often start with an EBNF description of a contract, which can be translated into protocol automata and then back into the detailed protocol contract notation. Henceforward, one would only adapt the generated detailed protocol contract notation by adding pre- and postconditions, invariants, and initial conditions as necessary.

Appendix D gives a full listing of the grammar of the detailed protocol contract notation.

4.5.3 Constraint Notation

Constraints refer to state properties of MONACO subcomponents (in general declared by their interface). In order to express such properties, we first declare subcomponents and then give constraints as Boolean propositions.

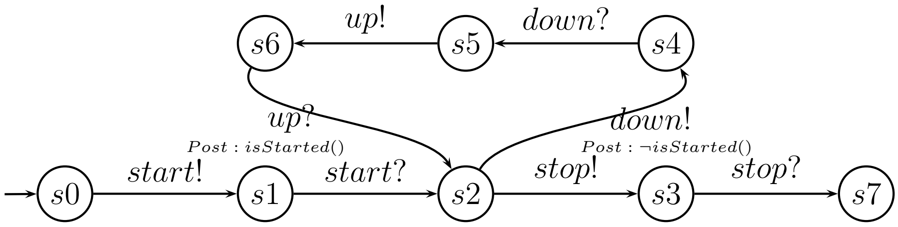


Figure 4.10: Protocol automaton resulting from Listing 4.5.

```
CONSTRAINT (ICooler cooler, IDriller driller)
     [NOT (driller.isStarted() AND NOT cooler.isCooling())]
```

Listing 4.6: Driller/Cooler constraint

Listing 4.6 shows the constraint defined above: the constraint affects the components `cooler` and `driller` implementing the interfaces `ICooler` and `IDriller` respectively. The condition states that it must never happen that the driller is started (`driller.isStarted()`) but the cooler is not cooling (`NOT cooler.isCooling`).

Appendix E gives a full listing of the grammar of constraints.

Chapter 5

Implementation Automaton

Chapter 4 introduced the notion of contracts, protocol automata, and constraints describing valid behavior of components. In this chapter we introduce means to represent component *implementations* as automata. In Chapter 6 then, we will see how our verification approach uses implementation automata to check them against contracts and constraints.

Section 5.1 introduces implementation automata, an automata formalism similar to protocol automata. Implementation automata reflect the actual sequence of calls in a MONACO component. In order to create the implementation automaton of a component, it is necessary to create sub-automata for every routine of the component (cf. Section 5.2). These automata will then be inserted into the automaton of the component's contract. The insertion of the routine automata into the parent component contract is called refinement and presented in Section 5.3. The implementation automaton thereby becomes an abstract representation of all possible execution paths of a component.

5.1 Automata Formalism

Implementation automata are similar to protocol automata presented in Section 4.2. Implementation automata represent the actual control flow within a component and contain all calls to subcomponents, as well as calls to local routines.

First, we introduce a formal description of a MONACO component.

Definition 5.1 *Let $C = \langle R, F, E, SC\rangle$ be the description of a component where the components R, F, E, SC have the following meaning:*

- *R is the set of routine symbols.*
- *F is the set of function symbols.*
- *E is the set of event symbols defined in the component.*
- *SC is the set of subcomponents. Let $sc \in SC$ be a subcomponent. The function $name(sc)$ then gives the name of the subcomponent, while $type(sc)$ gives the interface of the subcomponent. Recall that subcomponents can only be declared with interface types.*

Remark: We disregard parameters in the description of functions and routines. Parameters play a minor role in MONACO programs in general, and in the verification approach in particular, while disregarding parameters eases the description.

Based on the definition of components we introduce implementation automata.

Definition 5.2 *We call the LTS-based automata formalism for describing implementation details* implementation automata. *An implementation automaton is a quintuple $IA = \langle S, s^{init}, A, s^{final}, T\rangle$ describing an LTS with only a single initial state, a constrained set of actions and a final state:*

- *S is the set of states.*
- *$s^{init} \in S$ is the initial state.*
- *$A = R \times \{call, ret\} \cup SCR \times \{call, ret\} \cup \{\tau\}$ is the set of actions (alphabet). R is the set of routine symbols defined in the* MONACO *component (see above). SCR is the set of subcomponent routine symbols. That means let $sc \in SC$ be a subcomponent with $type(sc) = I_{sc} = \langle R_{sc}, F_{sc}, E_{sc}\rangle$ then $SCR = \bigcup_{sc\in SC} R_{sc}$. τ is the empty action representing an unconditional, immediate transition.*

- $s^{final} \in S$ *is the final state.*
- $T \subseteq S \times A \times S$ *is the transition relation.*

Remark: In the following, routine symbols of subcomponents are qualified with the name of the respective subcomponent *name*(*sc*). For example, consider a subcomponent `driller` of type `IDriller`. The symbol for the subcomponent's routine *start* would then be *driller.start*.

Additionally, two functions are introduced to represent conditions attached to states of the implementation automaton. In the following, conditions are logical propositions over all function symbols of subcomponents plus numerical and Boolean constants. That means that we use the function symbols from F_{sc} as logical variables. Functions with numerical return type can be used with relational operators and numerical constants. We allow the combination of logical expressions with the logical operators $\wedge$, $\vee$, and $\neg$. That means let $sc \in SC$ be a subcomponent with $type(sc) = I_{sc} = \langle R_{sc}, F_{sc}, E_{sc} \rangle$ then allowable function symbols are $\bigcup_{sc \in SC} F_{sc}$. We denote the set of all logical propositions over symbols $f \in \bigcup_{sc \in SC} F_{sc}$ as C.

The functions to represent conditions attached to states are:

- $CFC : S \rightarrow C$ is the function mapping states to control flow conditions. These conditions stem from control flow statements like **`IF`**, **`WHILE`** or **`WAIT`** and are valid at the associated states.
- $Post : S \rightarrow C$ is the function mapping states to postconditions. These postconditions stem from the contract of the component and need to be verified in the component implementation. For details on these postconditions, see Section 6.6.1.

Figure 5.1 shows the overall process of creating an implementation automaton: First, the automata of the routines are created. These automata are then inlined into the component's protocol automaton wherever a call to the respective routine is found. The automaton of a routine may even be inlined multiple times, if there is more than one call in the protocol automaton. Inlining routine calls is only possible because MONACO disallows recursive routine calls (Section 3.3.1). In the following, we show the construction process in detail.

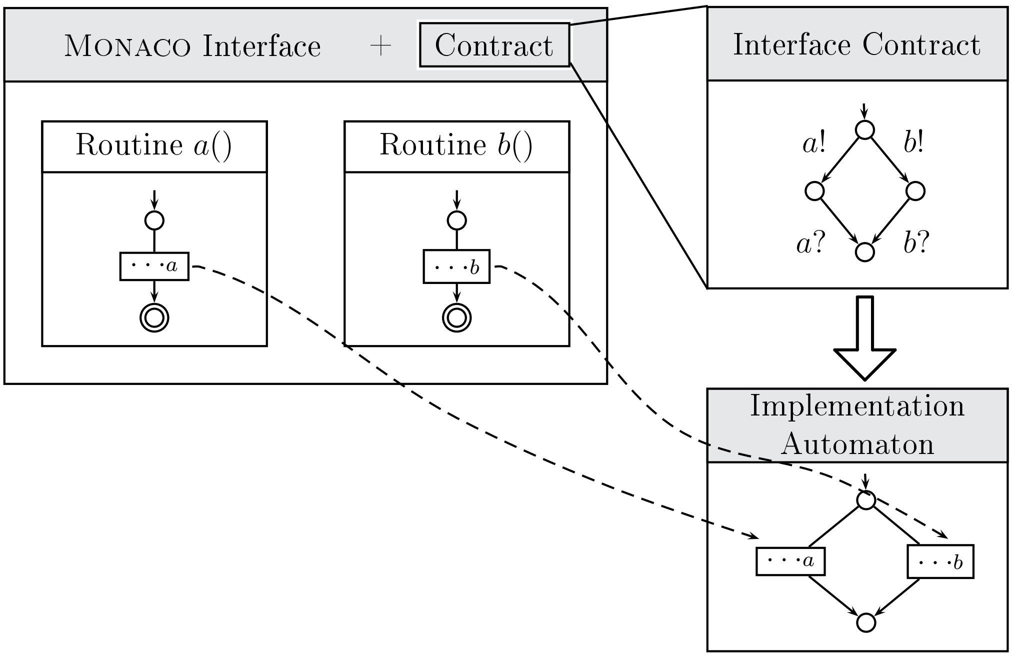


Figure 5.1: The full implementation automaton of a component is built from the implementation automata of its routines, inlined into the component's protocol automaton.

5.2 From MONACO to an Automaton

This section describes how an implementation automaton is created from an existing implementation of a MONACO component. We will start by first defining how routine calls are translated to implementation automata. Then, we will show concatenation of implementation automata to model a sequence of routine calls (or other statements). The last part of this section deals with MONACO control flow statements and their translation to implementation automata.

5.2.1 Routine Calls

Routine calls to subcomponents are the essential statements upon which we build implementation automata. The following code example shows a call to the routine `RoutineA` of the subcomponent `subc`.

```
subc.RoutineA();
```

Listing 5.1: Calling a **ROUTINE** of a subcomponent

Remark: Calls of component routines (contrary to subcomponent routines) are treated as if the statements of the routine were inlined at the location of the routine call.

As in protocol automata, routine calls are modeled by two transitions: the first transition models the call of the routine $(r, call)$, the second transition models the return of the routine call (r, ret).

Definition 5.3 *A* call *of a routine r of a subcomponent creates an implementation automaton P as follows:*

- $S_P = \{s, s', s''\}$ *is the set of states necessary to express a call. The state s is the state before the call, the state s' is the state during the call and the state s'' is the state after the call.*
- $s_P^{init} = s$ *is the state before the routine call.*
- $A_P = \{(subc.RoutineA, call), (subc.RoutineA, ret)\}$ *is the set of actions used in this implementation automaton.*
- $s_P^{final} = s''$ *is the state after the call of the routine.*
- $T_P = \{(s,(subc.RoutineA, call), s'),(s', (subc.RoutineA, ret), s'')\}$ *is the set of transitions between the states.*

Remark: If the called routine r is atomic (cf. Section 3.3.1) then the property $isAtomic(s')$ holds.

Figure 5.2 shows an implementation automaton that models such a simple routine call.

Remark: The presented notation of implementation automata only cares about routine calls to subcomponents and **WAIT**/**IF** statements (for knowledge extraction). Therefore all other statements (except for control flow statements) like assignment statements are ignored.

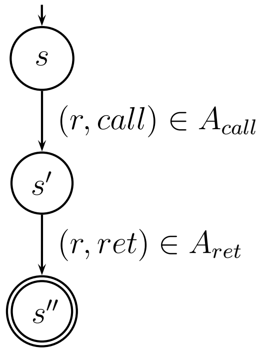


Figure 5.2: Implementation automaton for a simple routine call.

5.2.2 Statement Sequences

In imperative programming languages – MONACO is one of them – programs typically consist of statements that are executed in sequence. To reflect a sequence of routine call statements, implementation automata can be concatenated. The following code example shows the sequence of two routine calls.

```
subc.RoutineA();
subc.RoutineB();
```

Listing 5.2: Calling two **ROUTINE**s of a subcomponent

Statement sequences, such as two consecutive routine calls are generated by automaton concatenation. The concatenation simply merges the final state of the implementation automaton of the first statement with the initial state of the implementation automaton of the second statement.

Definition 5.4 *In general, the* concatenation *(sequential composition)* $P \circ Q$ *of two implementation automata* P *and* Q *is defined as follows:*

- $S_{P \circ Q} = S_P \cup S_Q \setminus s_Q^{init}$*. The set of states consists of the states of both implementation automata, without the initial state of the second automaton.*
- $s_{P \circ Q}^{init} = s_P^{init}$*. The initial state of the first automaton remains the initial state of the resulting automaton.*
- $A_{P \circ Q} = A_P \cup A_Q$*. The set of actions is the union of the actions of the two implementation automata.*

- $s^{final}_{P \circ Q} = s^{final}_{Q}$. *The final state of the second automaton remains the final state of the resulting automaton.*

- $T_{P \circ Q} = T_P \cup \{(s, a, s') \in T_Q \mid s \neq s^{init}_Q\} \cup \{(s^{final}_P, a, s') \mid (s^{init}_Q, a, s') \in T_Q\} \cup \{(s, a, s^{final}_P) \mid (s, a, s^{init}_Q) \in T_Q\}$. *The transitions in the concatenated implementation automaton consist of all transitions of the first automaton plus all transitions of the second automaton where transitions involving the initial state are bent over to the first automatons final state.*

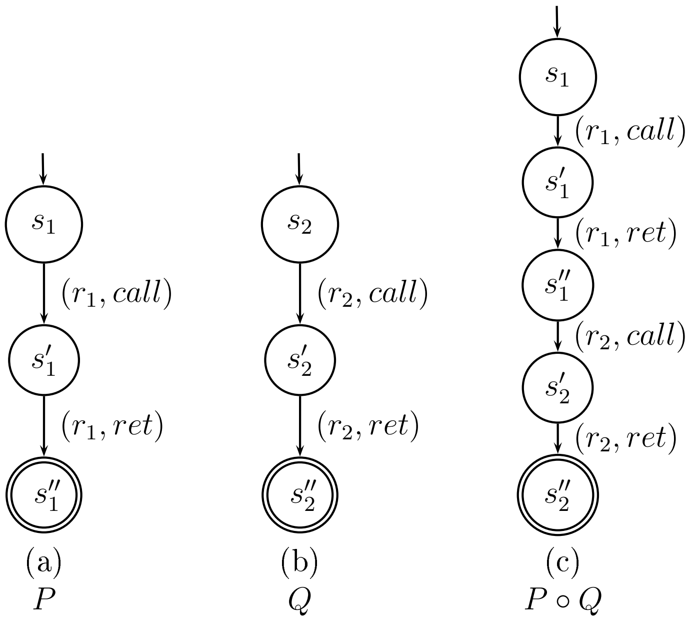


Figure 5.3: Implementation automata for simple routine calls ((a) and (b)) and the concatenation (c) of the two protocol automata.

Figure 5.3 (c) shows the concatenation of two automata. Note that $P \circ Q$ means that P is executed prior to the execution of Q.

5.2.3 Wait Statement

The **WAIT** statement ensures that a certain condition holds by suspending execution until the condition holds. Therefore we can use the condition in the implementation automata by adding this knowledge as a control flow condition to a new state s.

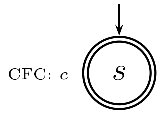


Figure 5.4: Implementation automaton for a wait statement.

Definition 5.5 *Adding new knowledge through the **WAIT** statement creates a single-state automaton as follows:*

- $S_{wait} = \{s\}$. *s is the single state of the implementation automaton.*
- $s_{wait}^{init} = s$. *The single state s is the initial state.*
- $A_{wait} = \emptyset$. *No actions are in this single state automaton.*
- $s_{wait}^{final} = s$. *The single state s is the final state.*
- $T_{wait} = \emptyset$. *There are no transitions in this automaton.*
- $CFCwait = \{(s, \{c\})\}$ *The CFC function for state s maps s to the condition of the **WAIT** statement.*

Figure 5.4 shows the implementation automaton resulting from a **WAIT** statement. Listing 5.3 shows a **WAIT** statement waiting for the function `isStarted` of the subcomponent `subc` to become true.

```
WAIT (subc.isStarted());
```

Listing 5.3: A MONACO **WAIT** statement waiting for a subcomponent.

5.2.4 Branch Statement

The MONACO **IF** statement can be used to branch the control flow. It allows one to specify any number of **IF** branches and one optional **ELSE** branch. Depending on the evaluation of the conditions, the control flow chooses one of the branches.

The semantics of the **IF** statement allows us to regard only a simple **IF** with an **ELSE** branch, since **ELSIF** branches can be seen as **ELSE** branches with an **IF** statement.

To create the implementation automaton for an **IF** statement, first the implementation automata of the **IF** and **ELSE** branch are built separately. If no **ELSE** branch exists, the implementation automaton for the non-existent branch consists of only a single state, being the initial and final state. The branching of the two implementation automata creates a common initial state as well as a common final state.

Definition 5.6 *The* branching automaton *of two implementation automata P and Q, where P describes the* ***IF*** *branch, and Q describes the* ***ELSE*** *branch of an* ***IF*** *statement, can be defined as follows:*

- $S_{P|Q} = S_P \cup S_Q \cup \{s_I, s_F\}$ *where s_I and s_F are new states.*
- $s_{P|Q}^{init} = s_I$ *is the new initial state. This state is where the automaton branches.*
- $A_{P|Q} = A_P \cup A_Q$ *is the combined set of actions.*
- $s_{P|Q}^{final} = s_F$*. The new state s_F is the new final state. This is where the branches merge.*
- $T_{P|Q} = T_P \cup T_Q \cup \{(s_Q^{final}, \tau, s_F), (s_P^{final}, \tau, s_F), (s_I, \tau, s_P^{init}), (s_I, \tau, s_Q^{init})\}$. *The set of transitions is extended by τ-transitions from the common initial state s_I to the initial states of P and Q. Similarly, τ-transitions from the final states of P and Q to the common final state s_F are added.*
- $CFC_{P|Q} = CFC_P \cup CFC_Q \cup \{(s_P^{init}, \{c\}), (s_Q^{init}, \{\neg c\})\}$ *is the control flow conditions function, where c is the branching condition.*

Figure 5.5 shows the automaton for an *IF* statement that has two branches. The conditions of the branches are as reflected in the automaton as control flow conditions (CFC) at the branching states.

5.2.5 Repetitions

The *repetition* of a block using MONACO's **WHILE** statement is done by first creating the implementation automaton P of the block that is to be repeated. The next step is to connect the final state of the block with a τ-transition to the initial state.

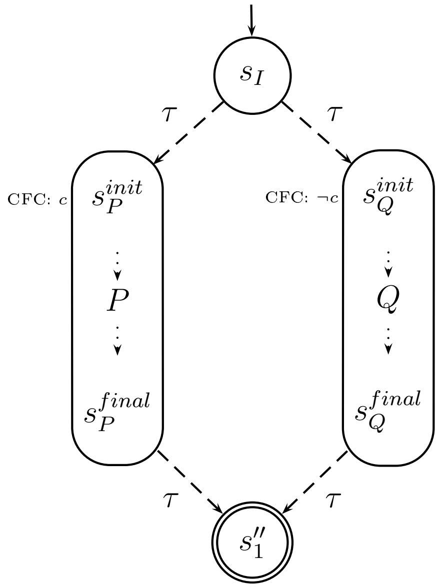


Figure 5.5: Implementation automaton for the Monaco *IF* statement. The automaton shows two branches.

Definition 5.7 *The implementation automaton for repeated execution of a code block P with* MONACO*'s* **WHILE** *statement is defined by:*

- $S_\circlearrowleft = S_P \cup \{s_I, s_F\}$ *is the set of states, where s_I and s_F are new states.*
- $s_\circlearrowleft^{init} = s_I$ *is the new initial state.*
- $A_\circlearrowleft = A_P$. *The set of actions remains the same.*
- $s_\circlearrowleft^{final} = s_F$ *is the single final state.*
- $T_\circlearrowleft = T_P \cup \{(s_I, \tau, s_F)\} \cup \{(s_I, \tau, s_P^{init})\} \cup \{(s_P^{final}, \tau, s_I)\}$. *Transitions are added from s_I to the old initial state and the new final state, as well as from the old final state to s_I.*
- $CFC_\circlearrowleft = CFC_P \cup \{(s_P^{init}, c)\} \cup \{(s_F, \neg c)\}$ *is the control flow condition function, where c is the* **WHILE** *condition.*

Figure 5.6 shows the implementation automaton resulting from the code in Listing 5.4.

```
WHILE c
BEGIN
  subc.RoutineA();
  subc.RoutineB();
END
```

Listing 5.4: **WHILE** statement

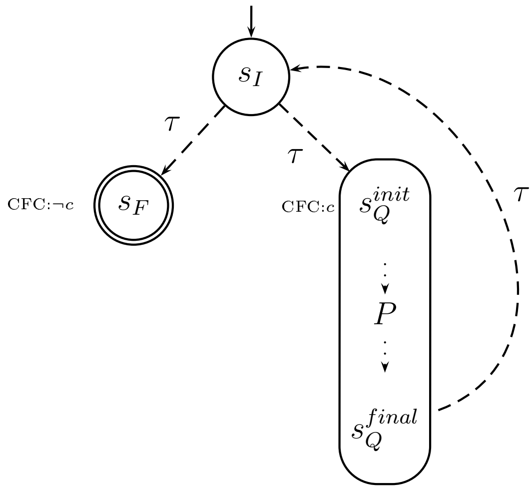


Figure 5.6: Implementation automaton for the Monaco $WHILE$ statement.

5.2.6 Parallel Statement

The **PARALLEL** statement is used to execute code in parallel. The following example shows the parallel execution of two routine calls.

```
PARALLEL
 subc.RoutineA(); // first parallel code block
||
 subc.RoutineB(); // second parallel code block
END
```

Listing 5.5: **PARALLEL** statement

The implementation automaton for the **PARALLEL** statement is created by *asynchronous composition* of the implementation automata of the parallel code blocks. We generate all possible interleavings of the parallel code blocks. The definition of asynchronous parallel composition is associative [Bie08], therefore $I_1 \parallel I_2 \parallel \cdots \parallel I_n$ can be constructed by first creating the parallel automaton $I_1 \parallel I_2$, and then using the resulting automaton to create $(I_1 \parallel I_2) \parallel I_3$. Therefore, we show the interleaving of two parallel blocks only.

Definition 5.8 *Let P, Q be two implementation automata, each representing a code block. The* asynchronous composition $P \parallel Q$ *of the two automata can be defined as:*

- $S_{P\parallel Q} = S_P \times S_Q$. *The set of states of two parallel automata is the Cartesian product of the sets of the two automata.*
- $s^{init}_{P\parallel Q} = (s^{init}_P, s^{init}_Q)$
- $A_{P\parallel Q} = A_P \cup A_Q$
- $s^{final}_{P\parallel Q} = (s^{final}_P, s^{final}_Q)$. *The final state is the pair of the final states of the two automata.*
- $T_{P\parallel Q} = \{((s_P, s_Q), a, (s'_P, s_Q)) \mid (s_P, a, s'_P) \in T_P\}$ $\cup \{((s_P, s_Q), a, (s_P, s'_Q)) \mid (s_Q, a, s'_Q) \in T_Q\}$. *Transitions in the parallel automaton describe the possible interleaving of the two automata.*

Figure 5.7 shows the parallel asynchronous composition of two automata P and Q. The figure clearly illustrates that by interleaving, any sequence of transitions is possible, as long as the sequence was possible in one of the original automata.

Interleaving of Atomic Calls

While the approach of interleaving all states of two parallel automata reflects the semantics of Monaco, it does not reflect the fact, that calls to atomic routines can not be interrupted (confer to Section 3.3.1). Therefore, if a state represents the state in an atomic routine call, then this state is not interleaved.

Definition 5.9 *We redefine the transition relation $T_{P\parallel Q}$ as follows:*

- $T_{P\parallel Q} = \{((s_P, s_Q), a, (s'_P, s_Q)) \mid (s_P, a, s'_P) \in T_P \wedge \neg isAtomic(s_Q)\}$ $\cup \{((s_P, s_Q), a, (s_P, s'_Q)) \mid (s_Q, a, s'_Q) \in T_Q \wedge \neg isAtomic(s_P)\}$.

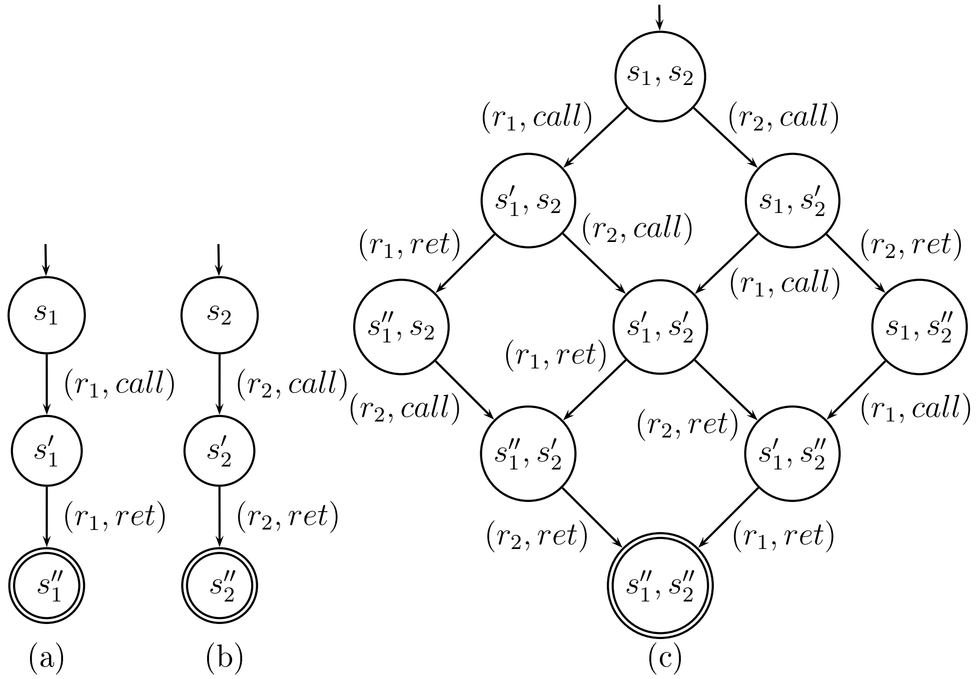


Figure 5.7: Implementation automaton for the Monaco $PARALLEL$ statement (c). The automaton shows the two parallel blocks P (a) and Q (b) being interleaved resulting in the automaton $P \parallel Q$.

Figure 5.8 shows the interleaving of calls to the routines $r1$ and $r2$, where the call to $r1$ is atomic.

5.2.7 Asynchronous Event Handling

Monaco offers an asynchronous event handling mechanism similar to the $try - catch$ construct of C/C++ style languages. Monaco's event handling mechanism allows one to guard the execution of a code block by an arbitrary condition. The semantics is, that the execution of the guarded block is terminated if the condition turns true. Execution then continues in the handler code.

Again, handling of events within a block is achieved by first creating the implementation automaton of the block that is guarded by the handler (P) and the implementation automaton of the handler code (Q). The next step is to create an event transition e from every state that is between a *call* and a *ret*-transition in the guarded block to the first state of the handler code. If

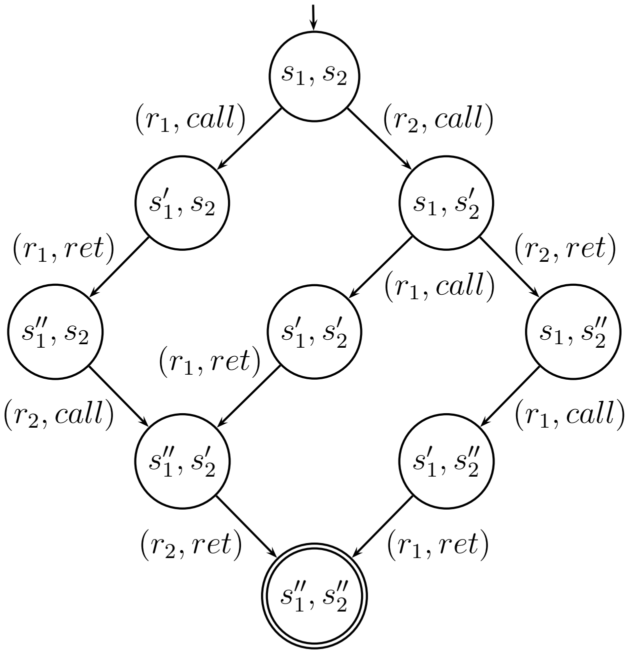


Figure 5.8: Implementation automaton for the Monaco $PARALLEL$ statement where routine r_1 is **ATOMIC** and thus $isAtomic(s_1')$ holds. In contrast to Figure 5.7 there is no interleaving of the state s_1'.

```
BEGIN
 subc.RoutineA(); // block guarded by the handler
ON subc.event
 subc.RoutineB(); // handler code
END
```

Listing 5.6: **ON** handler

the handler automaton is an empty automaton, transitions are created from any state of the guarded block to the final state. At the end of the handler block, execution continues after the guarded block.

Definition 5.10 *Adding an* event handler *automaton Q for an event condition c to an implementation automaton P is defined as follows:*

- $S_{P \rightsquigarrow Q} = S_P \cup S_Q$ *is the set of states.*
- $s^{init}_{P \rightsquigarrow Q} = s^{init}_P$. *The initial state of the guarded automaton remains is the initial state of the resulting automaton.*

- $A_{P \rightsquigarrow Q} = A_P \cup A_Q$ *is the set of actions.*
- $s^{final}_{P \rightsquigarrow Q} = s^{final}_P$. *The final state of the guarded automaton remains.*
- $T_{P \rightsquigarrow Q} = T_P \cup T_Q \cup \{(s, \tau, s^{init}_Q) \mid \exists s', r : (s', (r, call), s) \in T_P \wedge \neg isAtomic(s)\} \cup \{(s^{final}_Q, \tau, s^{final}_P)\}$. *Event transitions from all call-sites of non-atomic routines to the initial state of the handler automaton are added.*
- $CFC_{P \rightsquigarrow Q} = \{(s^{init}_Q, \{c\})\} \cup \{(s, \neg c) \mid s \in S_P \wedge s \neq s^{init}_P \wedge s \neq s^{final}_P\}$ *is the CFC function, where c is the condition of the* **ON** *handler (if such a condition exists). The condition is true in the initial state of the on handler and is false in the guarded block.*

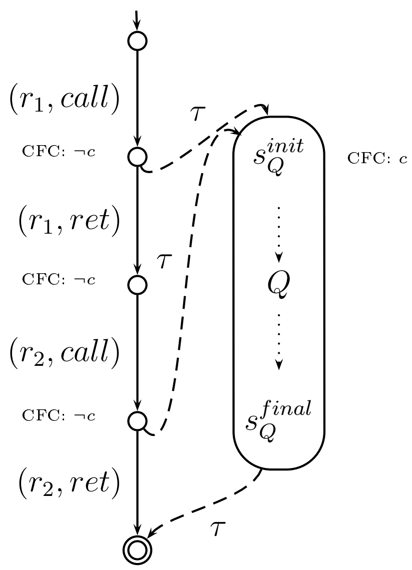


Figure 5.9: Implementation automaton for the Monaco event handling construct.

Figure 5.9 shows the implementation automata for a code block (show in Figure 5.10 and an event handler block and how the handler block is attached to the guarded code block.

Using the definitions above, we are able to create implementation automata for arbitrary MONACO code within a single routine. The automaton reflects the sequences of routine calls, routine returns and events that are possible in the respective MONACO code.

```
BEGIN
  r1();
  r2();
ON c
  // Q
END
```

Figure 5.10: Code for the event handling example in Figure 5.9.

5.3 Automata Refinement

Automata refinement describes the process of creating an implementation automaton for a component. This is done, by creating implementation automata for all routines of the component. These automata are then inlined into the protocol automaton of the component, wherever a call to the respective routine is found. Figure 5.1 gives an overview of this process. This way, the abstract description of the parent component (C, the protocol automaton of the component interface) is incrementally refined to a more concrete one (C', the implementation automaton of the component) [Sif01].

Definition 5.11 *We call the replacement of calls within a protocol automaton PA_C by the implementation automaton IA_r that models the implementation of the component's routine r the* refinement of PA_C by IA_r. *We denote this refinement $PA_C \lessdot IA_r$. Let $PA_C = P$, $IA_r = Q$, and for each call site of routine r, define the states $callStart_i, inCall_i, callRet_i \in S_P$ describing a call site i of routine r in P. The three states therefore are connected with the transitions $(callStart_i, (r, call), inCall_i)$ and $(inCall_i, (r, ret), callRet_i)$.*

The automaton resulting from inlining a routine call at call site i , $IA_{P \lessdot Q}$ is formally defined by

- $S_{P \lessdot Q} = (S_P \cup S_Q) \setminus \{inCall_i\}$. *The resulting set of states combines the two automata's states without the state modeling the call execution ($inCall_i$).*
- $s^{init}_{P \lessdot Q} = s^{init}_P$. *The initial state of P remains.*
- $A_{P \lessdot Q} = A_P \cup A_Q$
- $s^{final}_{P \lessdot Q} = s^{final}_P$. *The final state of P remains.*

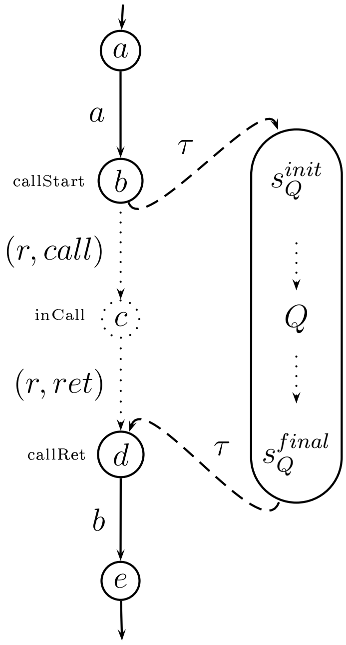


Figure 5.11: The refinement of the protocol automata P and with the implementation automaton Q of routine r inlines Q into P ($I_{P \lessdot Q}$) and removes the node of the original call.

- $T_{P \lessdot Q} = (T_P \cup T_Q) \setminus \{(s, a, s') \in T_P \mid s = inCall_i \vee s' = inCall_i\}$ $\cup \{(callStart_i, \tau, s_Q^{init}), (s_Q^{final}, \tau, callRet_i)\}$. *For the call site i, τ transitions to the initial state of Q, as well as τ transitions from the final states of Q to the the return of the call are added.*

Remark: At each call site, a separate copy of the implementation automaton of the routine is inlined, as we inline one call site after the other.

In other words, the refinement of a protocol automaton by the implementation automaton of a routine *inlines* a copy of the implementation automaton wherever there is a call to this routine in the protocol automaton. The resulting automaton is the basis for verification and semantic assistance presented in Chapter 6 and Chapter 7.

Figure 5.11 shows the refinement of the protocol automaton P by the implementation automaton of the routine Q. The implementation automaton is called Q, therefore the refinement can be denoted as $I_{P \lessdot Q}$.

Chapter 6

Verification Approach

This chapter presents the verification algorithm developed as a central part of this thesis. The results of this algorithm are the basis for the end-user assistance tools presented in Chapter 7.

Section 6.1 gives an overview of the approach. The description of the verification algorithm is split into 4 main parts. Section 6.2 introduces the basic verification algorithm that establishes a mapping between a component implementation and the protocol contracts of its subcomponents. Section 6.3 presents the operators chosen for the knowledge update between states in the implementation automaton. Section 6.4 introduces constraint checking, while Section 6.5 explains how unreachable states can be found. Finally, Section 6.6 presents how a component contract is checked against the component implementation.

6.1 Overview

The application of the verification algorithm is depicted in Figure 6.1. First, an automaton is created (as outlined in Chapter 5) which represents the implementation of a MONACO component with the control flow and the routine calls to its subcomponents (1). Then, a weak simulation relation is used to set up a mapping (3) between the states in the implementation automaton and the states in the protocol automata of the contracts (2) of the subcomponents. In the same step, the states of the implementation automata

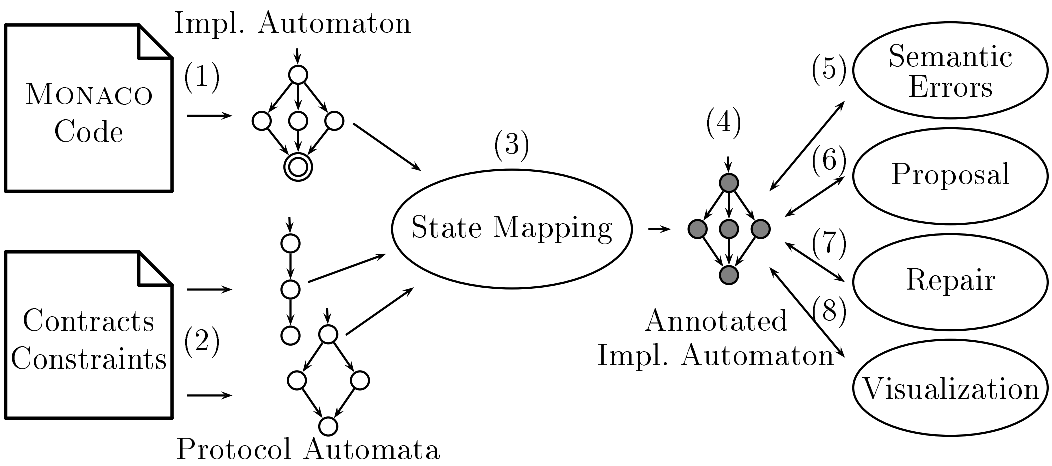


Figure 6.1: State mapping overview.

are associated with knowledge in the form of propositions derived from the propositions in the protocol automata and the conditional statements in the implementation. Finally, the state mapping and associated knowledge is used to verify constraints.

The annotated implementation automaton (4) is then used in various end-user support systems as follows:

- Reporting semantic errors (5) : The system gives feedback about violations of contracts and or constraints. The feedback is shown at the respective error positions in the editor.
- Proposing valid calls (6): Based on the contracts of the subcomponents and constraints between components the system proposes valid routine calls.
- Proposing semantic program repair (7): Component violating contracts or constraints can be changed such that the program complies with the contracts and constraints. This system gives proposals on which changes are necessary to repair a component.
- Visualizing component state (8): The system uses the state mapping results at a specific location in the code to visualize the state of the subcomponents at this exact location.

Those end-user support systems will be subject of Chapter 7.

6.2 State Mapping

This section introduces the state mapping algorithm for establishing a simulation relation between a component's implementation automaton and the protocol automata of its subcomponents. Section 6.2.1 introduces weak simulation relations. Section 6.2.2 discusses the principal approach and Section 6.2.3 presents the state mapping algorithm. Finally, Section 6.2.4 concludes with an example.

6.2.1 Weak Simulation

A *simulation* between automata describes that each transition in one automaton has a counterpart in the second automaton. The automata are said to have similar behavior (the second automaton may have more behavior).

A *weak simulation* [Bie08,Mil89] is a simulation disregarding unobservable internal events (τ-transitions).

Definition 6.1 *Let s_P, s_Q be states of the automata P and Q, then a weak simulation relation $\lesssim$ between these states is defined as follows: $s_P \lesssim s_Q \Leftrightarrow \forall a \in A_P \setminus \{\tau\}, s'_P \in S_P : (s_P \xrightarrow{\tau^* a} s'_P \Rightarrow \exists s'_Q \in S_Q : (s_Q \xrightarrow{\tau^* a} s'_Q \wedge s'_P \lesssim s'_Q))$ where the notation $s \xrightarrow{a} s'$ stands for $\exists (s, a, s') \in T$ and $s \xrightarrow{\tau^* a} s'$ stands for $s \xrightarrow{\tau} s_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} s_n \xrightarrow{a} s'$. An automaton Q weakly simulates an automaton P iff the initial state of Q weakly simulates the initial state of P: $s_P^{init} \lesssim s_Q^{init}$.*

Weak simulation is often used to verify an implementation against its specification. If $implementation \lesssim specification$ the implementation's behavior is a subset of the behavior allowed by the specification.

6.2.2 Approach

The weak simulation described above can be used to verify the implementation of a component against the sequencing constraints specified by the protocol automata of its subcomponents. In order to be able to describe the weak simulation between the implementation automaton and a protocol automaton of a subcomponent, we need to ignore all transitions resulting from

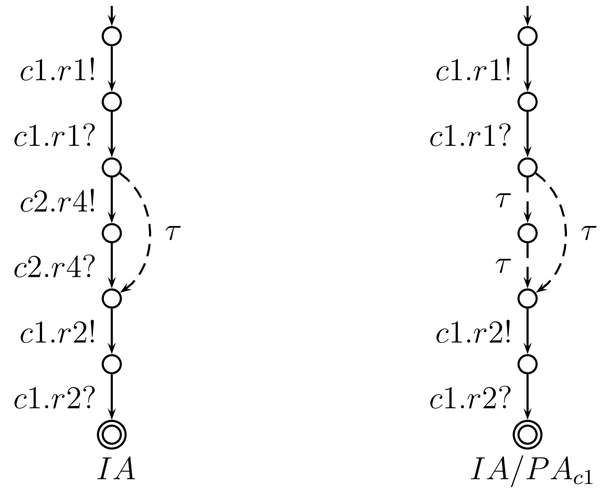


Figure 6.2: Statemapping projection of the implementation automaton on the protocol automaton of component $c1$ (PA_{c1}).

calls to other subcomponents. We simply replace these irrelevant transitions by τ-transitions and call this a projection of the implementation automaton on the protocol automaton of a specific component.

Definition 6.2 *We define the projection of an implementation automaton* $IA = \langle S_{IA}, s_{IA}^{init}, A_{IA}, s_{IA}^{final}, T_{IA} \rangle$ *on a protocol automaton* $PA = \langle S_{PA}, s_{PA}^{init}, A_{PA}, S_{PA}^{final}, T_{PA} \rangle$ *as an automaton* $IA/PA = \langle S_{IA}, s_{IA}^{init}, A_{PA}, s_{IA}^{final}, T_{IA/PA} \rangle$ *where* $T_{IA/PA} = \{(s, a, s') \in T_{IA} | a \in A_{PA}\} \cup \{(s, \tau, s') | (s, a, s') \in T_{IA} \wedge a \notin A_{PA}\}$.

This definition guarantees that all transitions in the resulting automaton are labeled with actions valid in the protocol automaton PA. The example in Figure 6.2 shows how projection replaces transitions involving subcomponents other than PA by τ-transitions.

In the state mapping algorithm we establish a weak simulation relation between the implementation automaton and each of the subcomponent protocol automata. Therefore a component C complies with the protocol automata of its subcomponents iff $\forall i : (IA/PA_i) \lesssim PA_i$.

Definition 6.3 *We define the mapping* $\mathcal{M}$ *of states of the implementation automaton to states of the subcomponent protocol automata as* $\mathcal{M} : S_{IA} \rightarrow$

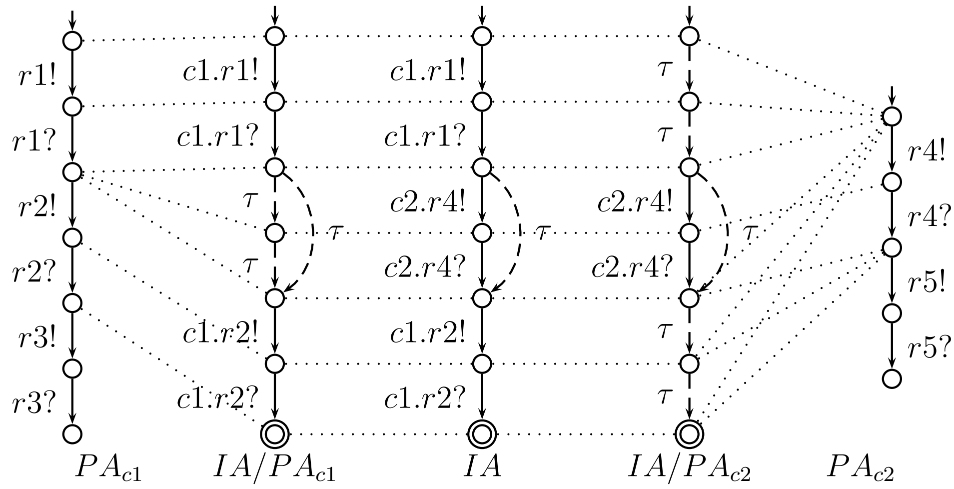


Figure 6.3: State mapping results with projection of the implementation automaton IA on the protocol automata of component $c1$ (PA_{c1}).

$\mathcal{P}(\times S_{PA_i})$. $\times S_{PA_i}$ denotes the cross product of the states of all subcomponents. Thus, this mapping relates a set of vectors of subcomponent states to a state of the implementation automaton. One such vector describes the state of all subcomponents. If multiple vectors are in the set, then the system can be in different states when execution reaches the state implementation automaton.

Let s_{IA} be the current state in IA and s_{PA_i} be the current state in the subcomponent protocol automaton PA_i. Assume, a transition $t_{IA} = (s_{IA}, a, s'_{IA}) \in T_{IA}, a \neq \tau$ exists in the implementation automaton. In order to have a weak simulation relation, a similar transition possibly reachable by intermediate τ-transitions $(s_{PA_i}, a, s'_{PA_i}) \in T_{PA_i}$ needs to exist in the corresponding protocol automaton. If so, a mapping between s'_{IA} and s'_{PA} is established: $\mathcal{M}(s'_{IA}) = \mathcal{M}(s'_{IA}) \cup \{(s_{PA_1}, \ldots, s'_{PA_i}, \ldots, s_{PA_n})\}$.

Figure 6.3 shows the result of the state mapping of an implementation automaton and two protocol automata for the subcomponents $c1$ and $c2$. For reasons of clarity, the projection automaton will be omitted from figures henceforward.

6.2.3 Algorithm

This section outlines the algorithm implementing the state mapping approach described above. The algorithm applies depth-first search (DFS) to find contract violations and annotates the states of the implementation automaton with mapping information.

Instead of establishing the weak simulation for each subcomponent separately, the algorithm does the projection on the fly. This allows the algorithm to establish the weak simulation in one depth-first search traversal of the implementation automaton. Moreover, rather than using the application stack by recursion, this algorithm is implemented iteratively, thus maintaining a separate stack of search positions. A *search position* holds a situation identified by a state in the implementation automaton and a corresponding state for each subcomponent protocol automaton. The search positions are connected through references to a predecessor search position, such that it is possible to follow the execution path leading to a certain state.

Definition 6.4 *A search position holds information about a state s of the implementation automaton as well as the mapped states of the subcomponent protocol automata. A search position therefore is a tuple $SP = \langle s, (t_1, \ldots, t_n)\rangle$, where*

- *$s \in S_{IA}$ is a state of the implementation automaton.*
- *$(t_1, \ldots, t_n)$ defines the subcomponent mapping, the active states in the subcomponent protocol automata. For each subcomponent there is one state in which this component is in this situation ($t_i \in S_{PA_i}$).*

A pseudo-code version of the algorithm is shown in Figure 6.4. The algorithm starts by assuming a mapping between the initial state of the implementation automaton s^{init} and the initial states of the subcomponent protocol automata t_i^{init} (line 1). While the search stack is not empty, the top search position is removed from the stack (line 3) and the (call- and return-) transitions leaving the implementation state s of the search position are verified to exist in the corresponding subcomponent protocol automaton. If such a transition exists, the mapping between the successor in IA and the successor in the subcomponent protocol automata is established (lines 12 and 19).

Input: implementation automaton, subcomponent protocol automata
Result: annotated implementation automaton, list of violations

1 $push(s^{init}, (t_1^{init}, ..., t_n^{init}))$
2 **while** *search positions on search stack* **do**
3     $\langle s, (t_1, \ldots, t_n) \rangle := pop()$
4   **foreach** a *such that* $\exists s' : (s, a, s') \in T_{IA}$ **do**
5   **if** $a \neq \tau \wedge \neg \exists i : (t_i, \tau^* a, t') \in T_{PA_i}$ **then**
6   violation detected at state s
7   **end**
8   **foreach** s' *such that* $(s, a, s') \in T_{IA}$ **do**
9   **if** $a = \tau$ **then**
10   **if** $(t_1, \ldots, t_n) \notin \mathcal{M}(s')$ **then**
11                     $push(s', (t_1, \ldots, t_n))$
12                     $\mathcal{M}(s') := \mathcal{M}(s') \cup \{(t_1, \ldots, t_n)\}$
13   **end**
14   **continue**
15   **end**
16   **foreach** t'_i *such that* $(t_i, \tau^* a, t'_i) \in T_{PA_i}$ **do**
17   **if** $(t_1, \ldots, t'_i, \ldots, t_n) \notin \mathcal{M}(s')$ **then**
18                     $push(s', (t_1, \ldots, t'_i, \ldots, t_n))$
19                     $\mathcal{M}(s') := \mathcal{M}(s') \cup \{(t_1, \ldots, t'_i, \ldots, t_n)\}$
20   **end**
21   **end**
22   **end**
23   **end**
24 **end**

Figure 6.4: DFS verification algorithm.

If the same mapping did not already exist, a new search position with the new successor of the transition in the implementation automaton and the new state mapping is pushed on the search stack (lines 11 and 18). If no such transition exists, a violation has been found (line 5). These steps are repeated until a mapping for each state has been found, or a violation is detected.

State mapping violations are due to invalid transitions in the implementation automaton. We can reconstruct a path leading to this violation using the search position chain. Each search position links to the search position causing this situation. Thus, the search positions can be seen as path lead-

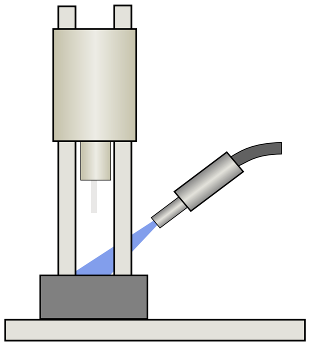


Figure 6.5: Driller and cooler component.

```
SUBCOMPONENTS
  c : ICooler;
  d : IDriller;
ROUTINE drill()
BEGIN
  c.start();
  d.start();
  WAIT d.rpmReached();
  d.down();
  d.up();
END
```

Listing 6.1: Partial implementation of the driller component.

ing from the initial state of the implementation automaton to the contract violation.

6.2.4 Example

Consider a driller machine like the one shown in Figure 6.5. The machine consists of two subcomponents, a driller and a cooler. Contracts exist for the interfaces of the subcomponents `IDriller` and `ICooler`, describing allowable usage patterns of the components. The driller machine could use its subcomponents like shown in Listing 6.1.

We can now apply the state mapping algorithm to the implementation automaton of the driller machine and the protocol automata of its subcomponents. The result of the state mapping algorithm is depicted in Figure 6.6.

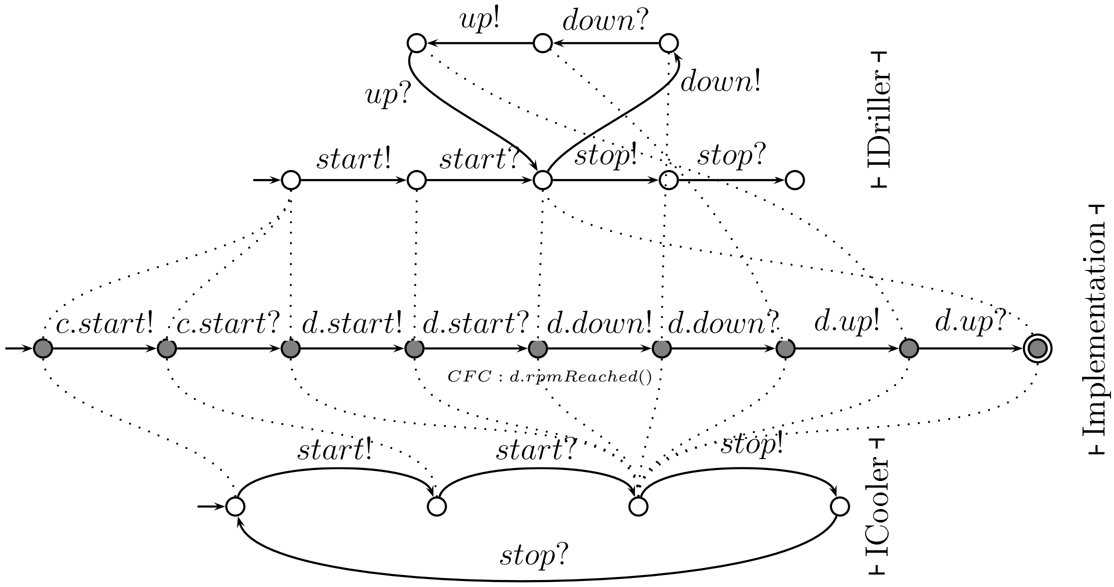


Figure 6.6: Result of the state mapping algorithm of the driller component.

The upper part shows the protocol automaton for the `IDriller` interface, the lower part shows the protocol automaton for the `ICooler` interface. In the center, the implementation automaton for the code in Listing 6.1 is shown. Dotted lines highlight the state mapping relation $\mathcal{M}$.

6.3 Knowledge Update

While the algorithm described in Figure 6.4 establishes a weak simulation relation, the propagation of knowledge in the implementation automaton has been omitted so far. This section will detail on situational knowledge, knowledge update and retraction, and we will present an extended state mapping algorithm propagating knowledge.

Situational knowledge is created from knowledge obtained from the protocol automata (see *Pre*, *Post*, and *Initial* functions in Section 4.3) and the implementation automaton (see *CFC* function in Section 5.1). Furthermore, we can use the function *Retract* from protocol automata to remove invalid knowledge. We use these propositions to annotate each reachable state of the implementation automaton with situational knowledge (similar to [Rei01]). The term situational knowledge refers to the fact, that a state in the implementation automaton may be reached through different paths in the implementation automaton, thus resulting in different knowledge and a different mapping of subcomponent states.

When a transition is taken, the situational knowledge of the source state is transferred to the target state of the transition. It is then updated with new information (from protocol automata) while keeping the situational knowledge consistent (i.e. the conjunction of all terms in the knowledge base must be satisfiable). We have adopted techniques introduced in artificial intelligence called belief update [KM91, HR99] and employed the SMT solver Yices [DdM06] to add and remove new information without introducing inconsistencies.

6.3.1 Knowledge Change Operators

We introduce a knowledge update operator (cf. Section 2.3.3) consistent with Winslett's standard semantics [Win90] (cf. Section 2.3.4). In contrast to belief revision, a belief update operation changes a knowledge base due to a change in the real world. The operation therefore may remove existing information from the knowledge base in order to keep the knowledge base consistent.

Definition 6.5 *Let K be the knowledge base consisting of a set of logical propositions $k \in K$ and c a logical conjunction describing new information about the world. Inv denotes the conjunction of invariant propositions. The knowledge update operator $\diamond$ is then defined as follows:*

$$K \diamond c = \{k \in K \mid \neg sameSym(k, c) \wedge sat(k \wedge c \wedge Inv)\} \cup c.$$

The predicate sameSym is true, iff the two propositions have at least one atom (symbol) in common. The predicate sat proves satisfiability of a proposition and is computed by an SMT solver.

Remark: We have chosen the SMT solver Yices [DdM06] as an efficient decision procedure for satisfiability of arbitrary formulas. Additionally it provides a simple input language which can be used in interactive mode.

Figure 6.7 shows the algorithm for the knowledge update in pseudo code. Each condition in the knowledge base is tested whether its symbols intersect with symbols contained in the new knowledge (line 3). If so, the condition is removed from the resulting knowledge base. Otherwise, the condition is tested whether it contradicts the new information and the invariants (line 5). If so, the condition is also removed from the resulting knowledge base. Finally the new information is added to the knowledge base (line 9).

Input: existing knowledge K, new information c, invariants Inv
Result: new knowledge base K'

1 $K' := K$
2 **foreach** $k \in K$ **do**
3   **if** $sameSym(k, c)$ **then**
4         $K' := K' \setminus \{k\}$
5   **elsif** $\neg sat(k \wedge c \wedge Inv))$ **then**
6         $K' := K' \setminus \{k\}$
7   **endif**
8 **end**
9 $K' := K' \cup \{c\}$

Figure 6.7: Pseudo code defining the knowledge update operator.

Similarly, an operator for information retraction can be defined. The semantics of retraction is that retracted knowledge can not be guaranteed to hold any longer. It therefore needs to be removed from the knowledge base.

Definition 6.6 *Let K be a knowledge base as above, and f a symbol to be retracted. The knowledge retraction operator $\blacklozenge$ is then defined as follows: $K \blacklozenge f = \{k \in K \mid \neg sameSym(k, f)\}$.*

Remark: Knowledge retraction differs from adding contradicting information, in that it does not generate additional information, but strictly removes any knowledge about certain symbols.

These knowledge operators are used in the state mapping algorithm to generate knowledge while establishing the weak simulation relation. The result of this state mapping algorithm including knowledge update is an annotated implementation automaton, where each reachable state is annotated with a list of situations. Each situation contains the subcomponent protocol automata mapping as well as a set of propositions known to be true in this situation. Section 6.3.3 gives a detailed definition of situations.

6.3.2 Example

In the following examples different cases for knowledge update in the state mapping algorithm are illustrated.

Adding Knowledge Based on Protocol Automata

Assume we have a subcomponent cooler of interface `ICooler` with the protocol automaton as defined in Figure 6.8, left column. The subcomponent is used as shown in Listing 6.2, the corresponding implementation automaton is depicted in Figure 6.8, right column. Dotted lines represent the state mapping relation.

```
SUBCOMPONENTS
  ICooler c;
  IDriller d;
ROUTINE main()
BEGIN
  c.start();
  ...
END
```

Listing 6.2: Example code generating knowledge from a protocol automaton.

The first part of Figure 6.8 shows the first mapping between the protocol automaton and the implementation automaton: the initial states are mapped and the mapping is annotated with the initial knowledge $K = \{\neg c.isCooling, \neg d.isStarted\}$. Next, the transition $c.start!$ in the implementation automaton is chosen as the only transition from the current (initial) state in the implementation automaton. The same transition (though without the subcomponent prefix $c.$) exists in the protocol automaton for the `ICooler` subcomponent c. Therefore, a mapping between these two successor states is established, the knowledge is not yet changed (since postcondition information is added to the knowledge as soon as the state holding the postcondition is left). The knowledge associated with this new mapping therefore remains $K = \{\neg c.isCooling, \neg d.isStarted\}$.

Finally, the next transition $c.start?$ is taken and its counterpart in the protocol automaton is followed. The postcondition of the state in the protocol

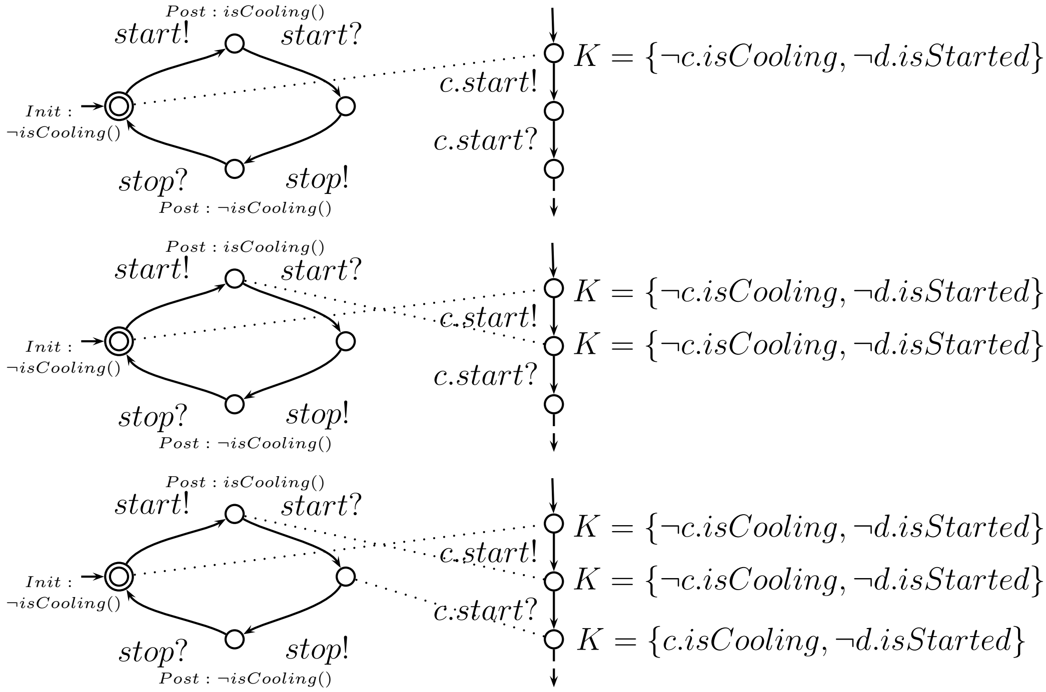


Figure 6.8: Verification process: state mapping and knowledge generation from protocol automaton.

automaton is used to update the current knowledge. Thereby, the proposition $\neg c.isCooling$ is removed because it shares symbol *isCooling* with the new proposition $c.isCooling$. Finally the new proposition is added to the knowledge base giving $K = \{c.isCooling, \neg d.isStarted\}$. Repeated execution of the code can lead to new mappings of implementation states to the same states in the protocol automaton (even with different knowledge). Similarly, one state in the implementation automaton can be mapped to multiple states in the protocol automaton (possibly with different knowledge per mapping).

Adding Knowledge Based on WAIT / IF

This example illustrates how information from the implementation automaton is used in the verification process and how preconditions are verified. Figure 6.9 shows the implementation automaton for the code snippet in Listing 6.3, where the system waits for the driller component to have reached full speed, before the driller lowers.

```
BEGIN
  ...
  WAIT d.rpmReached();
  d.down();
  ...
END
```

Listing 6.3: Example code generating knowledge from the implementation automaton.

Figure 6.9 shows the protocol automaton of the `ICooler` subcomponent on the left, the protocol automaton of the `IDriller` subcomponent on the right, and the implementation automaton for the code snippet in the center. Assume, the knowledge at the state of the CFC condition is $K = \{c.isStarted, d.isStarted\}$. Before the transition $d.down!$ is taken in the implementation automaton and the protocol automaton for the `IDriller` interface of the subcomponent `d`, the knowledge is immediately updated with the CFC condition. The temporary knowledge therefore is $K = \{c.isStarted, d.isStarted, d.rpmReached\}$.

Next, the precondition of the successor state in the protocol automaton of `IDriller` is verified. Since $K \wedge \neg Pre$ is not satisfiable, the precondition $d.rpmReached$ is satisfied, the transition $d.down!$ is taken, and the mapping between the two successor nodes is established. The knowledge in the second implementation state is then $K = \{c.isStarted, d.isStarted\}$. It lacks the function symbol $d.rpmReached$, because this knowledge can no longer be guaranteed as it does not stem from a contract guarantee, but from a **`WAIT`** statement, and the system may have changed due to the routine call.

Remark: The SMT solver can only show satisfiability or unsatisfiability of formulas. Therefore, a precondition is fulfilled, if its negation is unsatisfiable under a certain knowledge. It does not suffice to show that the precondition and the knowledge are satisfiable.

Retracted Knowledge Based on Protocol Automata

This example shows how retraction of information from existing knowledge can be used. Figure 6.10 shows the section of the implementation automaton

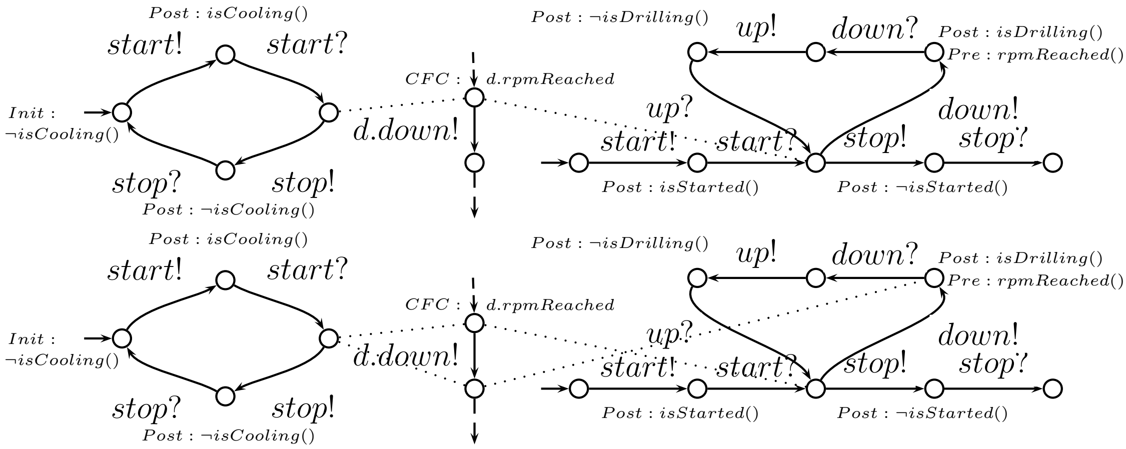


Figure 6.9: Verification process: state mapping and knowledge generation from implementation protocol.

for the code snippet in Listing 6.4, where the system starts the close movement of a cylinder, waits for the cylinder to be closed, and then stops the movement.

The protocol automaton for the interface `ICylinder` of the cylinder subcomponent states, that as soon as a movement is started, no conclusion about the state of the subcomponent can be drawn ($Retract : isOpen, isClosed$). Assume, we have the knowledge $K = \{cyl.isOpen\}$ when the verification process arrives at the first state of the implementation automaton shown in Figure 6.10.

```
BEGIN
  ...
  cyl.startClose();
  WAIT cyl.isClosed();
  cyl.stop();
  ...
END
```

Listing 6.4: Example code showing retraction of knowledge.

When the transition $cyl.startClose!$ is taken, it leads to a state in the protocol automaton which is annotated with a set of symbols to retract. All propositions involving retracted symbols are removed from the knowledge. In the example, the proposition $cyl.isOpen$ is removed from the knowledge and an empty knowledge remains ($K = \{\}$). In the next step, temporarily

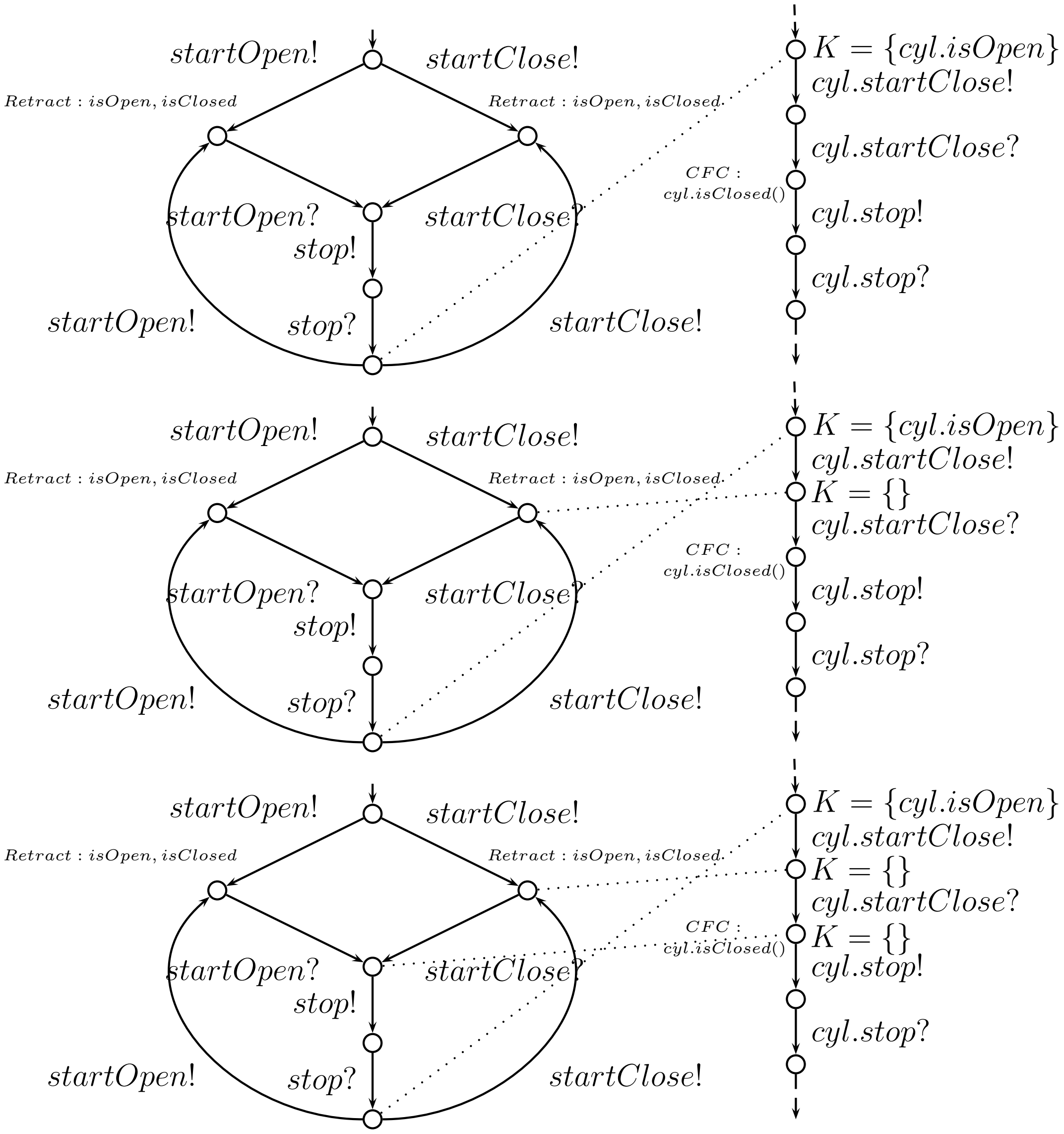


Figure 6.10: Verification process: state mapping and knowledge retraction.

new knowledge is added from the CFC condition of the implementation. The temporary knowledge thus is $K = \{cyl.isClosed\}$. This knowledge stemming from the CFC condition would only be used if there were a precondition in the protocol automaton. Since there is no precondition, the CFC condition is not used and the empty knowledge $K = \{\}$ remains.

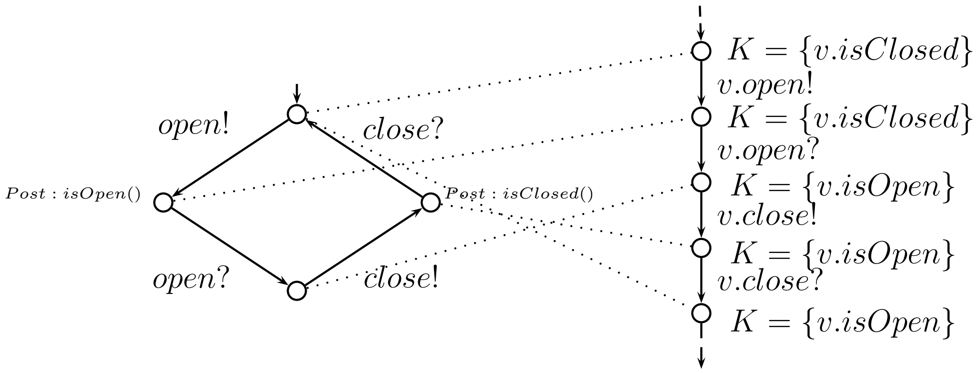


Figure 6.11: Verification process: knowledge update with invariants.

Knowledge Update with Invariants

This example shows how invariants influence the result of knowledge update. Assume, we have a valve subcomponent which can be opened and closed (atomic routines *open*() and *close*()). The functions declared in the interface of the valve subcomponent are *isOpen* and *isClosed* which can never be true simultaneously. We describe this dependence using the invariant $\neg(isOpen \wedge isClosed)$(the preceding $\neg$ can be read as *never*).

Listing 6.5 shows an example code which uses the valve subcomponent. The corresponding implementation automaton is shown in Figure 6.11. The interesting part of the knowledge update is in the third state of the implementation automaton. The knowledge from the previous state is $K = \{v.isClosed\}$ and the new information from the postcondition is $v.isOpen$. The knowledge update step generates the final knowledge $K = \{v.isOpen\}$ by removing $v.isClosed$ because $v.isClosed \wedge v.isOpen \wedge \neg(isOpen \wedge isClosed)$ is not satisfiable.

```
BEGIN
  ...
  v.open();
  WAIT 1000;
  v.close();
  ...
END
```

Listing 6.5: Example code for knowledge update with invariants.

6.3.3 Algorithm

Figure 6.12 gives the full pseudo code of the state mapping algorithm, including knowledge update operations. The following adaptations need to be made to the state mapping algorithm:

- The mapping is changed to map sets of situations to implementation states. A situation is a tuple $Situation = \langle (t_1, \ldots, t_n), K \rangle$. The new mapping therefore is $\mathcal{M} : S_{IA} \rightarrow \mathcal{P}(Situation)$. New situations are added to the mapping as they occur (lines 14 and 25).
- A new element K representing the set of propositions on subcomponents valid in the implementation automaton is added to the search position. Therefore, it is now defined as $SP = \langle s, (t_1, \ldots, t_n), K \rangle$.
- Unsatisfiability of control flow conditions need to be checked (line 9).
- New information from control flow conditions needs to be added to the knowledge (line 10).
- Knowledge needs to be retracted, if specified in the protocol automata (line 8).
- Knowledge from **`WAIT`** statements needs to be retracted as soon as it is not valid any more (line 21). This is the case, as soon as the next non-atomic routine is called after the **`WAIT`** statement.
- New information from the protocol automata needs to be added to the knowledge (line 19).
- Constraints need to be checked whenever a new mapping is generated (line 22). This is the subject of the next section.

6.4 Constraint Checking

Constraints are checked in the state propagation algorithm in every situation encountered (line 22). A constraint is satisfied, iff the current knowledge contradicts the negated constraints, i.e., if there is no possibility that the current knowledge and an invalid state (as described by constraints) coincide.

Input: implementation automaton, subcomponent protocol automata
Result: annotated implementation automaton, list of violations

1 $push(s^{init}, (t_1^{init}, ..., t_n^{init}), \bigcup_i Initial(PA_i))$
2 **while** *search positions on search stack* **do**
3     $\langle s, (t_1, \ldots, t_n), K\rangle := pop()$
4   **foreach** *a such that* $\exists s' : (s, a, s') \in T_{IA}$ **do**
5   **if** $a \neq \tau \wedge \neg\exists i : (t_i, \tau^* a, t') \in T_{PA_i}$ **then** violation detected
6   **foreach** s' *such that* $(s, a, s') \in T_{IA}$ **do**
7   **let** PA_i such that $\exists t' : (t_i, \tau^* a, t') \in T_{PA_i}$
8             $K' := K \blacklozenge Retract(t_i)$
9   **if** $\neg sat(K' \wedge CFC(s'))$ **then continue** with line 6
10             $K' := K' \diamond CFC(s)$
11   **if** $a = \tau$ **then**
12   **if** *mapping is new* **then**
13                     $push(s', (t_1, \ldots, t_n), K')$
14                     $\mathcal{M}(s') := \mathcal{M}(s') \cup \{((t_1, \ldots, t_n), K')\}$
15   **end**
16   **continue** with line 6
17   **end**
18   **foreach** t'_i *such that* $(t_i, \tau^* a, t'_i) \in T_{PA_i}$ **do**
19                 $K'' := K' \diamond Post(t_i)$
20   **if** $sat(K'' \wedge Inv \wedge \neg Pre(t'_i))$ **then** violation detected
21                 $K'' := K'' \blacklozenge invalid\ WAIT\ knowledge$
22   **if** $sat(K'' \wedge Inv \wedge \neg Constr)$ **then** violation detected
23   **if** *mapping is new* **then**
24                     $push(s', (t_1, \ldots, t'_i, \ldots, t_n), K'')$
25                     $\mathcal{M}(s') := \mathcal{M}(s') \cup \{((t_1, \ldots, t'_i, \ldots, t_n), K'')\}$
26   **end**
27   **end**
28   **end**
29   **end**
30 **end**

Figure 6.12: DFS verification algorithm with knowledge update.

```
CONSTRAINT (ICooler cooler, IDriller driller)
    [NOT (driller.isStarted() AND NOT cooler.isCooling())]
```

Listing 6.6: Driller/Cooler constraint.

```
1 (define cooler_isCooling::bool)
2 (define driller_isStarted::bool)
3 (define constraint::bool (not (and driller_isStarted (not cooler_isCooling))))
4 (assert cooler_isCooling)
5 (assert driller_isStarted)
6 (assert (not constraint))
7 (check)
```

Listing 6.7: Yices input for checking a constraint.

Definition 6.7 *More formally, a constraint is* violated*, iff* $sat((\neg Constr) \wedge invariants \wedge knowledge)$

In order to solve this SAT problem, again the SMT solver Yices [DdM06] is used. The satisfiability problem is translated into the input language of the SMT solver, which in turn returns either satisfiable or unsatisfiable.

Assume we have to check the constraint in Listing 6.6. The situational knowledge is $cooler.isCooling() \wedge driller.isStarted()$ and there are no invariants. The SAT problem for checking the constraint reads as follows:

$sat(\neg\neg(driller.isStarted() \wedge \neg cooler.isCooling()) \wedge cooler.isCooling() \wedge driller.isStarted())$

The input for Yices for this satisfiability problem is listed in Listing 6.7. Lines 1 and 2 declare the two boolean symbols used in the constraint and the knowledge. Line 3 defines the constraint and lines 4 and 5 assert the knowledge. Line 6 asserts that the constraint is violated, which needs to be unsatisfiable. The last line executes the *check* command which checks the previous commands for satisfiability and either returns *sat* or *unsat*.

The given SMT problem is unsatisfiable, since $\neg cooler.isCooling()$ and $cooler.isCooling()$ can not hold simultaneously. Hence, the *check* command returns *unsat* and the constraint is not violated. If the SMT solver reported satisfiability of the problem, we would have found an instance of constraint violation.

6.5 Reachability Analysis

Reachability analysis aims at finding code which is unreachable and thus is either superfluous or flawed. Unreachable code is also often called *dead code*. It seems natural to extend static analysis to find such code, since the state mapping algorithm already does most of the static analysis needed. What remains to do for a reachability analysis is to analyze the results of the state mapping algorithm.

The analysis is done by checking the states in the annotated implementation automaton having a control flow condition (from **IF** or **WHILE** statements). Each such state must have at least one situation in which the control flow condition is established, in order to be executable. If there is no situation in which the condition holds, an unreachable state has been found.

```
1  BEGIN
2    cooler.start()
3    driller.start();
4    WAIT driller.rpmReached();
5
6    IF NOT cooler.isStarted() THEN
7    BEGIN // unreachable code block
8      cooler.start();
9    END
10
11   driller.down();
12   driller.up();
13   ...
14 END
```

Listing 6.8: Unreachable code.

Listing 6.8 shows a Monaco code block containing unreachable code. The unreachable code is the block starting at line 7. It is caused by the preceding **IF** statement which has a condition that will never be true due to the postcondition knowledge gathered by the call to `cooler.start()` in line 2. The result of the verification and reachability analysis is shown in Figure 6.13.

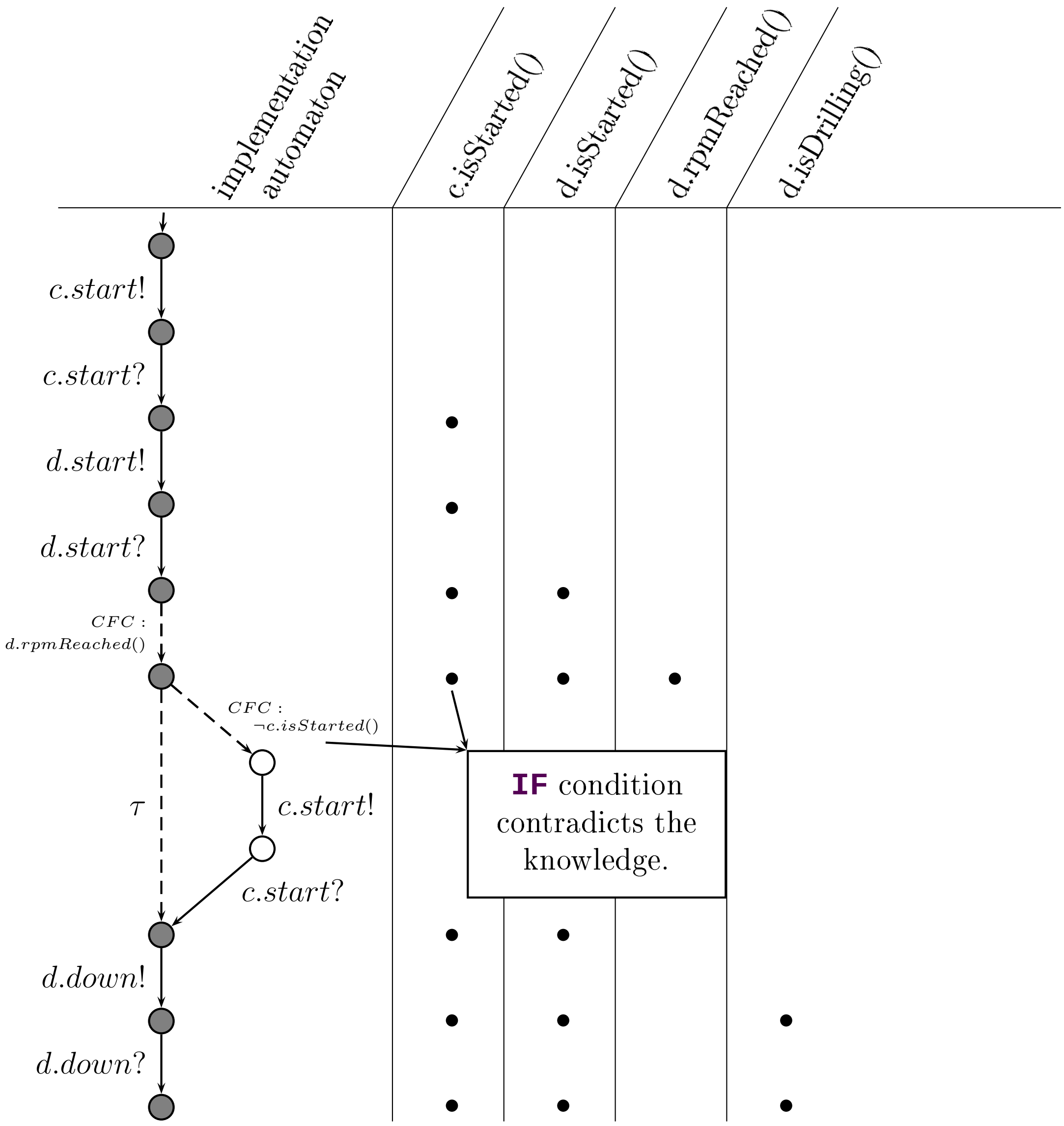


Figure 6.13: Unreachable code due to unsatisfiable **IF** condition.

6.6 Checking Component Contracts

Recall from Section 3.2.3 that the component structure forms a strict hierarchy. The verification for one component relies on the contracts of its subcomponents and assumes that its routines are called as required by its own contract. This kind of reasoning is referred to as assume-guarantee reasoning [HMP01]. To be sound, the component implementation has to guarantee that it fulfills the postconditions specified in its contract. This will be outlined in the following.

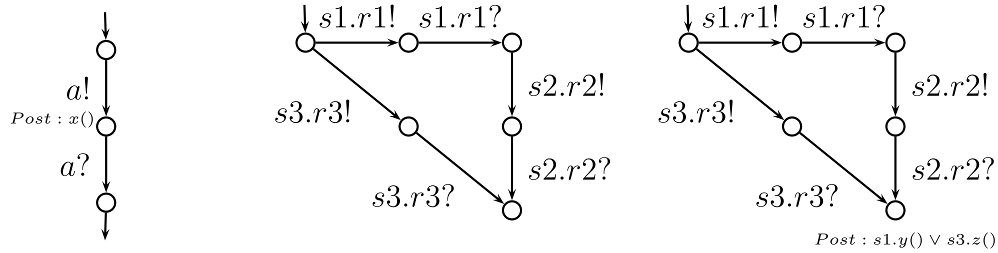
6.6.1 Checking Component Postconditions

As described above, a component has to guarantee, that it fulfills the postconditions specified in its contract. We can check this by adding the postconditions of the contract of a component to the implementation automaton, when the component routines are inlined (see Section 5.3).

The only problem is, that the postconditions of the component are stated in terms of function symbols of the component itself, while the conditions used in the knowledge update procedures are stated in terms of the function symbols of the subcomponents. This can be solved by analyzing the code of the functions used in these postconditions. These functions essentially return aggregated states of their subcomponents. Thus, they consist of a single `RETURN` statement with a condition composed of subcomponent function symbols. This exact condition is then used within the new postcondition.

Figure 6.14 gives an overview of the process: the postcondition x of the routine call a (left) is added to the implementation automaton of the routine $a()$. Since the symbol x is a function of the component and not of one of its subcomponents, the contents of the function x are used. Let's assume the code of the function x is `RETURN s1.y() OR s3.z()`. The postcondition is then $s1.y \vee s3.z$ and added to the last state of the implementation automaton (right).

In the state mapping algorithm the actual check for compliance with the postconditions of the component's contract has to be done after line 19 (see algorithm in Figure 6.15). The check verifies that the knowledge (K'') implies the parent postcondition. If this check fails, the component does not fulfill



Protocol Automaton IComponent | Implementation Automaton for a() | Implementation Automaton for a()

Figure 6.14: Postconditions of the component's contract are transferred to their implementation automaton. The postcondition is thereby stated in terms of subcomponent function symbols.

its contract.

20 **foreach** t_i' *such that* $(t_i, \tau^* a, t_i') \in T_{PA_i}$ **do**
21     $K'' := K' \diamond Post(t_i)$
22   **if** $sat(K'' \wedge Inv \wedge \neg Post(s'))$ **then** violation detected
23   **if** $sat(K'' \wedge Inv \wedge \neg Pre(t_i'))$ **then** violation detected
24     $K'' := K'' \blacklozenge invalid\ WAIT\ knowledge$
25   **if** $sat(K'' \wedge Inv \wedge \neg Constr)$ **then** violation detected
26   **if** *mapping is new* **then**
27         $push(s', (t_1, \ldots, t_i', \ldots, t_n), K'')$
28         $\mathcal{M}(s') = \mathcal{M}(s') \cup \{((t_1, \ldots, t_i', \ldots, t_n), K'')\}$
29   **end**
30 **end**

Figure 6.15: Part of the DFS verification algorithm. Line 22 checks whether the postcondition of the component's contract is fulfilled.

6.6.2 Checking Unchanged State Properties

The check described in Section 6.6.1 above guarantees that postconditions are fulfilled. In addition to postconditions, there is a second assumption that we use when updating knowledge in the state mapping algorithm: knowledge gained from postconditions remains true, until it is invalidated (by another postcondition, or by knowledge retraction).

We can verify this assumption by checking that the knowledge at component routine calls only change state properties of the component, if these changes are specified in the routine's postcondition.

Chapter 7

Semantic Assistance

> "Syntax is what you see,
> semantics is what you
> have to find out."
> - *Anonymous*

This chapter introduces techniques to assist end users in programming. These techniques exploit the verification approach as presented in Chapter 6. Section 7.1 presents an algorithm for searching for proposals that suggest how a MONACO program can be legally extended or modified at a specific location. Section 7.1.2 shows how these proposals can be used to build interactive end-user support tools. The same algorithm forms the basis for the semantic program repair approach (Section 7.2), which fixes components that are invalid with respect to their contracts. The last section of this chapter (Section 7.3) presents a program visualization tool that can show and animate the state of components during programming.

The term *Semantic Assistance* is derived from the Eclipse term *content assist*, a facility that provides programmers with proposals about what words the user could type in the current context (cf. Section 2.1). Our approach is to use syntactic information plus semantic knowledge (contracts) to give correct proposals (with respect to the contracts) instead of only taking syntactic information into consideration. As introduced in Section 6.1, Semantic Assistance tools are based on information gathered from checking components against contracts and constraints of their subcomponents. That means that it relies on the state mapping and knowledge deduction process as presented in Chapter 6. The resulting annotated implementation automaton is used by the tools presented in this chapter to give proposals to the end user, which are not only syntactically correct, but also semantically valid with respect to the semantics given by protocol contracts and constraints.

7.1 Search for Proposals

This section introduces a search procedure for finding valid routine calls for a certain position in the source code. The procedure finds those states of the implementation automaton that correspond to the given position in the code. These states are then used to find a set of valid routine calls with which the call sequence up to this point can be continued.

Definition 7.1 *Let's assume, that the state s is the implementation state corresponding to a specific location in the source code, where we want to compute which routine calls are allowed to occur next. We define the set of* valid routine calls *as:*
$VC = \{r \mid \exists i \forall \langle (s_{PA_1}, \ldots, s_{PA_n}), K\rangle \in \mathcal{M}(s) : \exists s' : (s_{PA_i}, \tau^*(r, call), s') \in T_{PA_i} \wedge \neg sat(((K \blacklozenge Retract(s_{PA_i})) \diamond Post(s')) \wedge Inv \wedge \neg Constr)$
$\wedge \neg sat(((K \blacklozenge Retract(s_{PA_i})) \diamond Post(s')) \wedge Inv \wedge \neg Pre(s'))\}$.

This means, that all routines are valid routines, where

1. a protocol contract PA_i allows us to call the routine in any situation $\langle (s_{PA_1}, \ldots, s_{PA_n}), K\rangle$ associated with the given implementation state s:
   $\exists i \forall \langle (s_{PA_1}, \ldots, s_{PA_n}), K\rangle \in \mathcal{M}(s) : \exists s' : (s_{PA_i}, \tau^*(r, call), s') \in T_{PA_i}$
   (for an example see Section 7.1.1)
2. the call does not violate any constraints $\neg sat(K' \wedge Inv \wedge \neg Constr)$ where K' is the updated knowledge $((K' \blacklozenge Retract(s_{PA_i})) \diamond Post(s'))$
3. the call does not violate a precondition $\neg sat(K' \wedge Inv \wedge \neg Pre(s'))$.

In essence, it can be regarded as simulating one step in the state mapping algorithm starting at the mapping of the implementation state s. Note, that it does not consider further steps, such that it does not recognize errors which appear subsequently as a result of a proposed routine call.

If more than one state corresponds to the code position for which the set of valid routine calls is to be calculated, the intersection of the valid routine calls of the respective states is the result. We need to use the intersection, because valid routine calls should be valid in any possible execution path leading to the code position.

7.1.1 Examples

In the following, examples will illustrate the search for proposals in various situations.

Valid Routine Calls at a Single State

Figure 7.1 shows an example of the application of the search for valid routine calls after the code in Listing 7.1. In this example, we want to find out, how we can proceed at the state s in the implementation automaton (center). The protocol automata of the subcomponents *cooler* and *driller* are depicted to the left and the right, respectively. The dotted lines show the state mapping relations between the states of the implementation automaton and the states of the subcomponent protocol automata.

Valid routine calls at the implementation automaton state s are the routine symbols at call-transitions leaving the protocol automata states mapped to s. In our example, the routines *c.stop* and *d.start* would be valid. These transitions are marked bold in Figure 7.1.

```
BEGIN
  c.start();
  <>
END
```

Listing 7.1: Example code for valid routine calls.

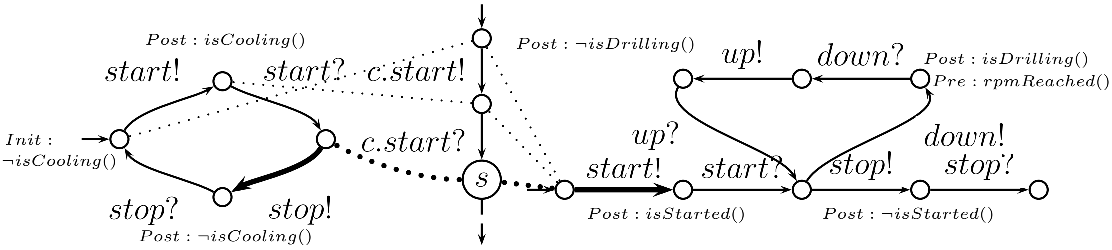


Figure 7.1: Finding valid routine calls.

```
BEGIN
  c.start();
  d.start();
  IF f.pieceAtDriller() THEN BEGIN
    WAIT d.rpmReached();
    d.down();
  END
  <>
END
```

Listing 7.2: Example code for valid routine calls.

Valid Routine Calls with Multiple Situations

When multiple different situations can be found for a certain position in the code, we have to use the intersection of the valid routine calls at each situation. Listing 7.2 shows a code sample where different situations occur at the cursor position. In this example, three subcomponents exist: a cooler and a driller subcomponent as in the previous example, and a feeding component transporting workpieces to the driller.

The corresponding implementation automaton is shown in Figure 7.2 (top). It shows the state mapping result at state s in the implementation automaton (dotted lines). Due to the two branches of the **IF** statement, two different situations emerge:

- Situation 1 with knowledge
  $K = \{c.isStarted, d.isStarted, \neg f.pieceAtDriller\}$
- Situation 2 with knowledge
  $K = \{c.isStarted, d.isStarted, d.isDrilling\}$

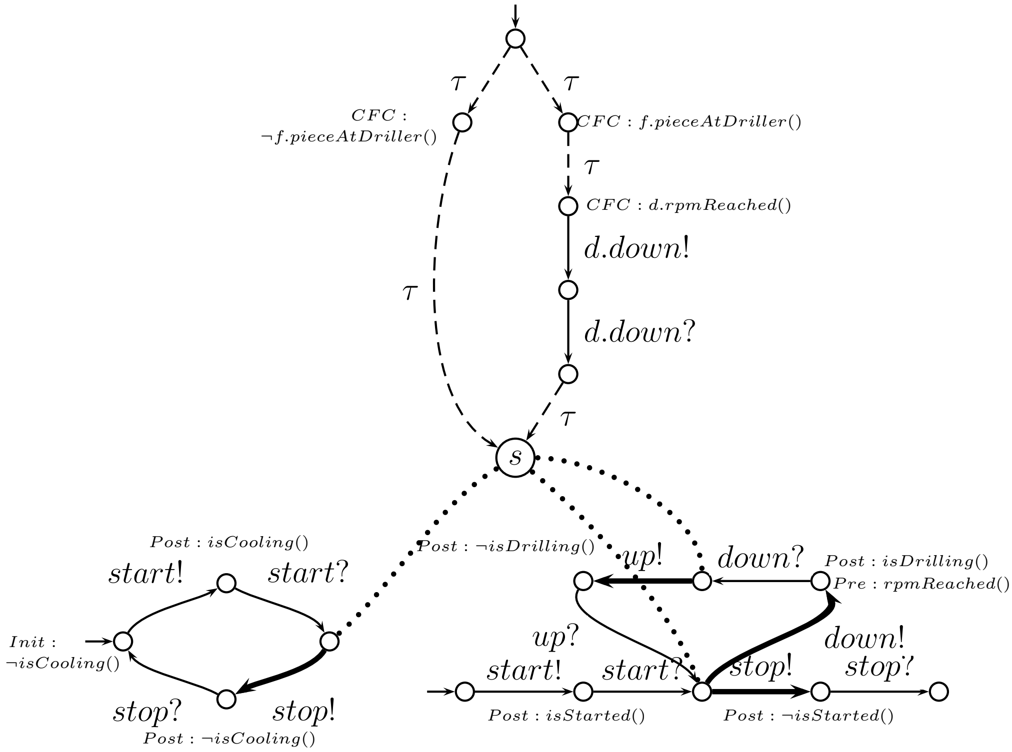


Figure 7.2: Finding valid routine calls.

The two situations do not only differ in the associated knowledge, but also in the mapped states of the driller protocol automaton (bottom right). The first situation (in which the **IF** branch was not taken) is mapped to the state directly after the return of the routine *start*. The second situation is mapped to the state between the return of routine *down* and the call of routine *up*.

Valid routine calls for this example per situation would then be:

- Situation 1: *d.stop*, *d.down*
- Situation 2: *d.up*

Since the intersection of these sets of valid routine calls is empty, no routines can be proposed at this position. Yet, guarded proposals can be made, which check for the active situation by proposing an **IF** statement before each of the routines. Guarded proposals in this example are as follows:

- Situation 1:
```
IF NOT f.pieceAtDriller() THEN d.stop();
IF NOT f.pieceAtDriller() THEN d.down();
```

- Situation 2:
```
IF d.isDrilling() THEN d.up();
```

Note, that the conditions of the guarded proposals are just the tests for the different situations. Guarded proposals are not yet implemented in the prototype implementation of Semantic Assistance.

7.1.2 Interactive Assistance

The functionality described above can be used to enhance existing code proposal facilities. In the following, three interactive tools for semantic end-user assistance are presented. All tools propose valid routine calls at a selected code position.

Semantic Assist Popup

Figure 7.3 shows the proposal popup of the Semantic Assistance implementation. While the popup presents all syntactically valid routines and functions of the subcomponent driller, it highlights those routines which do not violate contracts or constraints.

driller.down() and *driller.stop*() are valid calls at the cursor position, while *driller.start*() and *driller.up*() are invalid and therefore crossed out. Still, also the invalid calls are shown in the popup menu and can even be selected and inserted. This is because a program must be allowed to violate its contracts temporarily during editing. After editing, the program is checked again before it is downloaded to the machine. By crossing out the invalid calls we at least indicate to the end user that a call to these routines is invalid here. Note that calls to functions are always possible.

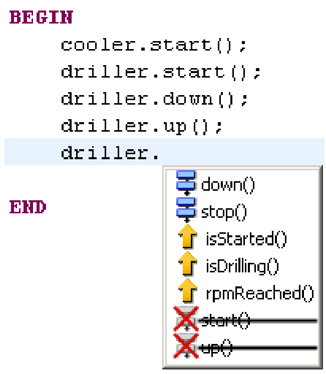


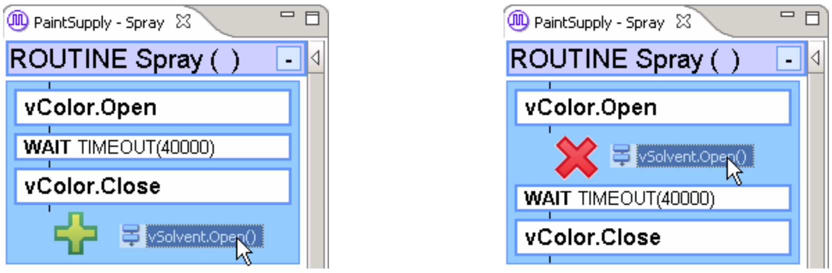
Figure 7.3: Semantic Assistance popup window showing valid routines and semantically invalid routines (crossed out).

Drag-and-drop Assistance

The Monaco visual editor allows a user to insert routine calls by drag and drop. For every component in the program there is a sidebar menu listing all possible routine calls to this component. The user can select a call from this menu and drag it into the code. While he moves the mouse cursor over statements the positions where the selected call can be dropped are highlighted. Valid positions are highlighted by a green plus sign (Figure 7.4(a)), while invalid positions are marked by a red cross (Figure 7.4(b)). The state information obtained from contracts and constraints is used to find the positions where a call can be dropped legally. Note, that it is again possible to drop a call also at an illegal position, thus violating the contracts of the program temporarily.

Outline Highlighting

The Eclipse outline view shows all routines valid at the selected code position. We have customized the outline view to show all routines that can be called at the selected code position according to the contracts. Figure 7.5 shows a screen shot of the outline view and the visual editor with a selected code position. The code position selected is between two statements (highlighted by a black rectangle), and according to the contracts, only one routine



(a) Call allowed.

(b) Call disallowed.

Figure 7.4: Drag-and-drop assistance in the visual editor. Figure (a) shows that it is possible to insert the call *vSolvent.Open()* at the selected location while (b) shows that it is not possible to insert it at another location.

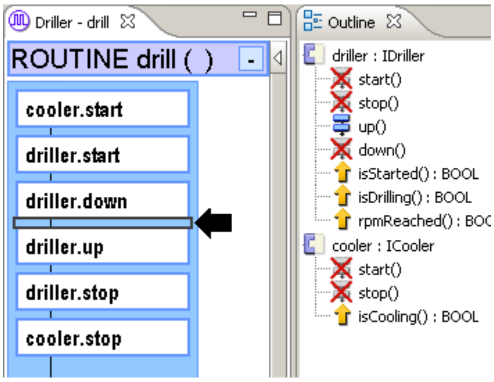
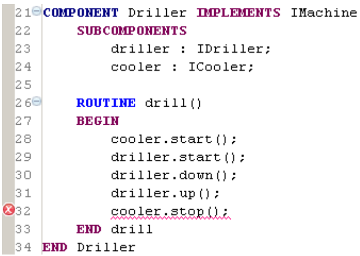


Figure 7.5: Semantic Assistance showing valid routines in the outline view.

(*driller.up*) is valid there. All other routines are crossed out.

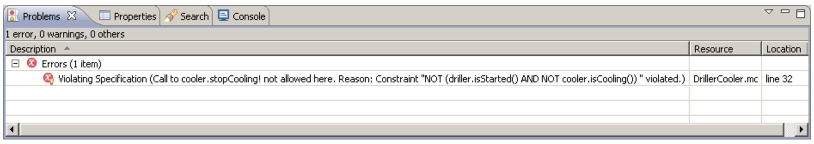
7.2 Program Repair

Semantic errors cannot be fully eliminated by the tools presented above. A user might need to make temporary changes to a program, turning the program invalid. These semantic errors are indicated in the textual editor by red underline and an error marker at the left margin. Similarly, these errors



(a) Semantic error in text editor.

(b) Semantic error in visual editor



(c) Semantic error in the MONACO problems view

Figure 7.6: Semantic error in the MONACO text editor and the MONACO visual editor. The error shown here is due to a constraint violation. Details on the error are presented in the MONACO problems view.

are also shown in the visual editor, where a light bulb marks an error which can be resolved by the procedures presented in this chapter. Additionally, semantic errors are shown in the Eclipse problems view.

Program repair is about changing a program containing a semantic error such that the change removes the contract violations. Figure 7.6 shows the different visualizations for semantic errors. Figure 7.12 in Section 7.2.3 shows the resulting repair proposals and the repaired program.

The goal of program repair is to recover from semantic errors by offering a list of program change proposals from which the developer can choose. Those proposals are based on the semantically invalid program and the contracts. Selecting any of the proposals will make the resulting program semantically valid. If a program contains more than one semantic error, the program repair algorithm might need to be applied multiple times.

7.2.1 Goals

The goals of the program repair algorithm are to provide program change proposals that:

1. do not introduce new errors,
2. remove existing semantic errors,
3. make as few changes as possible,
4. are as close as possible to the error location.

Goals 1 and 2 are necessary goals, while goal 3 can be quantified in terms of number of changes and an associated weight per change operation. The weight of one program change proposal is the sum of the weighted change operations and can be used to rank different program change proposals and find those that make minimal changes while still fulfilling goals 1 and 2 (lower weight ranked higher). Goal 4 aims for local changes that an end user programmer can comprehend by looking at the code where the error occurred, without having to search through several routines.

7.2.2 Repair Strategies

The repair strategies of the program repair algorithm differ based on the type of semantic error. The types of semantic errors that we can find are as follows:

1. Invalid call sequence: the sequence of routine calls in the program violates the sequences allowed by the protocol automaton of a subcomponent.
2. Condition violated
   (a) Precondition violated: a subcomponent routine is called without having the precondition of this call established.
   (b) Constraint violated: a call to a subcomponent generates knowledge that violates one or more constraints.

(c) Parent postcondition violated: at the end of a routine, the postcondition of the routine in the component's contract is not fulfilled.

Error type 1 (invalid call sequence)

The semantic errors of type 1 boil down to an invalid routine call due to a missing transition in the protocol automaton. These errors can be fixed by changing the transitions in the implementation automaton. The following repair strategies can therefore be chosen:

- Insert a routine call which is valid in the contracts (weight: 2)
- Remove a routine call (weight: 3)
- Move a routine call to some other position (total weight: 1)

Error type 2 (condition violated)

Semantic errors of type 2 can only be fixed by creating new knowledge, such that the condition currently violated is fulfilled when the repair proposals are applied. A repair proposal therefore can consist of the following repair strategies:

- Insert calls establishing the necessary condition (weight: 2).
- Remove a routine call (weight: 3).
- Insert a **`WAIT`** statement, if the code position is within a parallel context or the violated condition is a precondition which can not be established by a postcondition of a routine (weight: 1.4).
- Insert an **`IF`** statement, if there is at least one situation in which the violated condition is satisfiable (weight: 3).

  Remark: If there was no situation in which the condition can be satisfied, there is no use in adding an **`IF`** statement, since it would only make the error location unreachable.

```
1 BEGIN
2   c.start();
3   d.start();
4   WAIT d.rpmReached();
5   d.down();
6   d.up();
7   c.stop();
8   d.stop();
9 END
```

Figure 7.7: MONACO code with semantic error due to constraint violation.

The weights have been chosen such that the goals stated above are met as good as possible. We assume, that certain mistakes are more common than others, therefore the repair proposals for these mistakes have a lower weight. Severe changes, like removing a routine call have the highest weight (3), since we can assume that an end user would not add an unnecessary routine call, but rather add it at an inappropriate location. Thus, moving a routine call has the least weight (1). Adding a routine call (without removing the same call at another location) has an intermediate weight (2).

7.2.3 Algorithm

The program repair algorithm uses bounded depth first search to find change proposals. In every step of the depth first search, all repair strategies (insert call, remove call, ...) are consulted to repair the program. As soon as a sequence of change actions has been found, that locally repairs the program, this set of changes is added as a new proposal to the result. A search path is no longer followed, if the depth has reached a certain limit, or the total weight of the changes has exceeded a maximum weight.

In order to illustrate the algorithm, we will demonstrate it by means of an example. The code with the semantic error can be found in Listing 7.7. Assume, we have a constraint defining that the cooler must not be stopped while the driller is started. The semantic error then is in line 7, where the cooler is stopped, before the driller.

Figure 7.8 shows the part of the implementation automaton containing the semantic error. The algorithm starts to search for program repair propos-

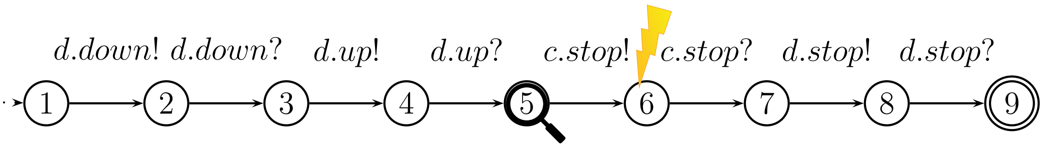


Figure 7.8: Program repair example.

als at the state directly before the statement where the violation was detected. In our example, this is state 5, directly before the transition *c.stop*!.

A fragment of the search tree is shown in Figure 7.9. For clarity, the insertion of **`WAIT`** or **`IF`** statements has been omitted as possible repair actions in Figure 7.9, because they do not lead to valid repairs in this particular example. Dashed edges indicate continuation of the search, while check marks label nodes with valid repair proposals. The latter nodes also contain a number denoting the total weight of the proposal.

We will take a look at one of the search paths, specifically, at the search path having the minimal total weight. This path is highlighted in Figure 7.9. The search procedure starts at the root of the search graph and reasons about changes to the implementation automaton. The first strategy consulted, is the strategy for adding routine calls. This strategy looks at the states mapped to the current state in the implementation automaton (state 5) and looks for legal continuation routine calls. In state 5, calls to the routines *d.down* and *d.stop* are valid according to the contracts. No calls to the subcomponent cooler are valid at this position, though.

The search procedure adds new branches (first *d.down*, then *d.stop*) to the search tree. We continue at the highlighted path in the search tree: now, the edge *d.stop* is followed, we add new virtual states connected by the transitions *d.stop*! and *d.stop*? to the implementation automaton. All the knowledge update and checking steps as conducted in the state mapping algorithm are performed, such that a virtual state mapping for the new states exists. Figure 7.10 shows the new virtual states.

The new terminal virtual state V_2 is used as the starting point for the next level in the search algorithm. Again, all repair strategies are consulted. We continue with the strategy following the highlighted path in Figure 7.9. This strategy is called *consume* and does not add any new virtual nodes to the implementation automaton, but marks the routine call following the insertion point of the virtual branch in the automaton as consumed. Doing so

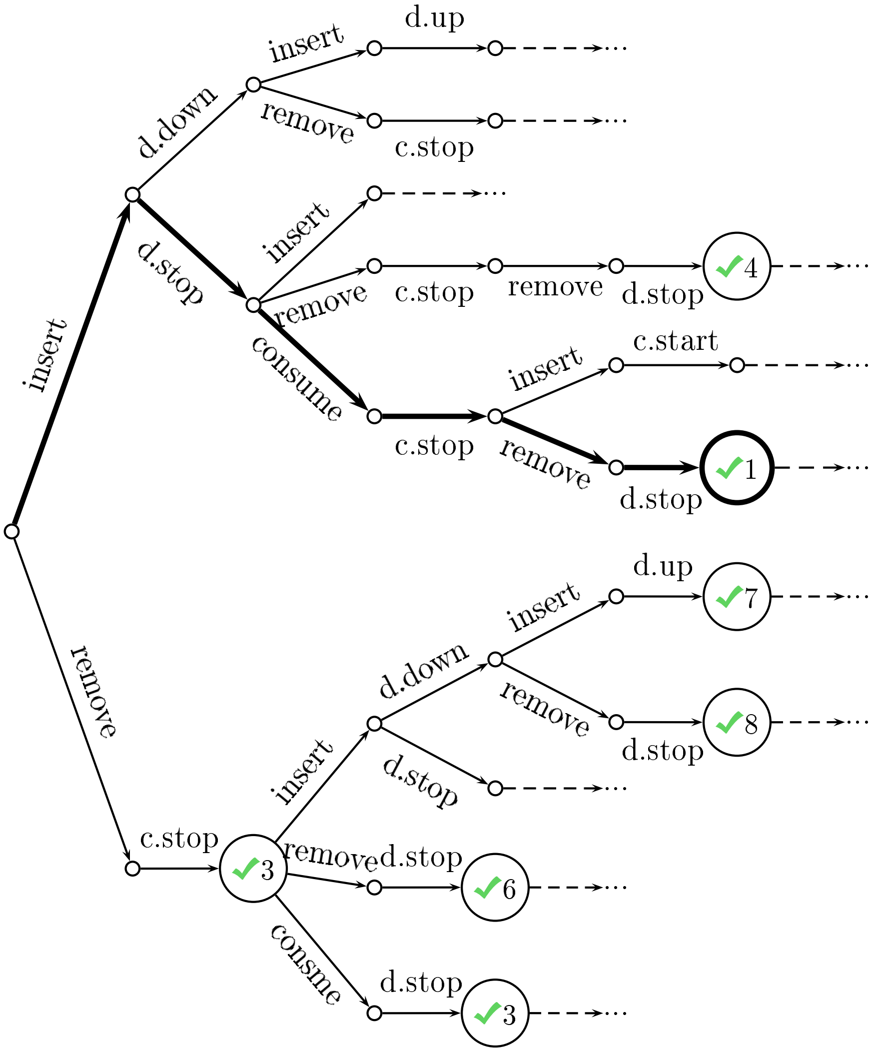


Figure 7.9: Program repair algorithm search tree.

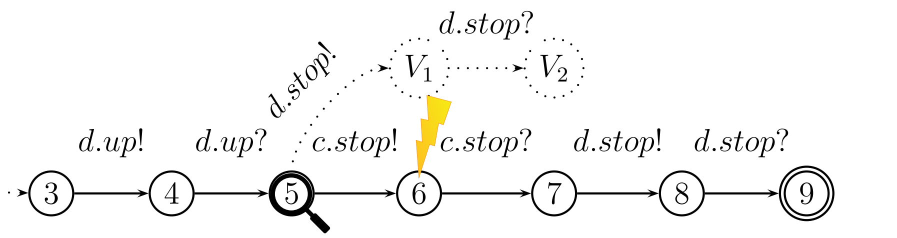


Figure 7.10: Program repair example with virtual states after one step.

is only possible if this call does not lead to any contract violations in the new virtual mapping. In the example, the call *c.stop* is consumed and a virtual

Repair Proposal	Total Weight
insert *d.stop*, remove *c.stop*, remove *d.stop*	4
insert *d.stop*, skip *c.stop*, remove *d.stop*	1
remove *c.stop*	3
remove *c.stop*, insert *d.down*, insert *d.up*	7
remove *c.stop*, insert *d.down*, remove *d.stop*	7
remove *c.stop*, remove *d.stop*	6
remove *c.stop*, skip *d.stop*	3

Figure 7.11: Repair proposals and their weight, ordered by appearance in the search tree.

state mapping and knowledge update is established for the consumed nodes.

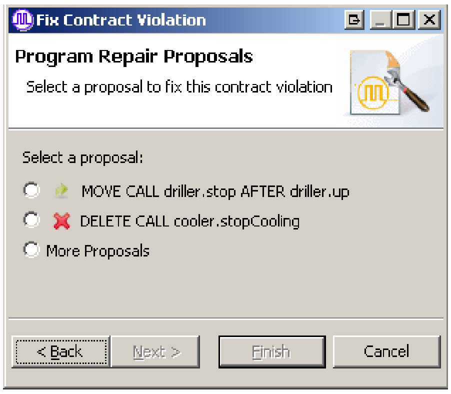
Next, again all repair strategies are consulted. After the insertion strategy was executed, the remove strategy removes the routine call *d.stop*. The resulting set of changes (insert *d.stop* before *c.stop* and remove the existing *d.stop* after *c.stop*) is a valid repair proposal. The question remains, how the algorithm detects whether a certain path in the search tree is a valid repair proposal.

Recognizing Valid Repair Proposals

After each application of a repair strategy, the algorithm checks whether the resulting virtual state mapping can be continued with the rest of the implementation automaton. Since a full state mapping application in every node of the search tree would take too long, only a fixed number of state mapping steps are performed. If no violations are found within these steps, the path leading to this node in the search tree is assumed to be a valid repair proposal.

Prioritizing Repair Proposals

In the example presented, several valid repair proposals have been identified. In order to present only the most adequate proposals to the programmer, these proposals need to be sorted. We use the total weight of a proposal based on the sum of the individual weights of the repair strategies.



(a) Program repair proposals wizard showing proposals for the example.

(b) Result of program repair.

Figure 7.12: Program repair wizard proposing repair actions with minimal impact.

Remark: Although a move strategy has been introduced it is not an explicit strategy, rather a consequence of an insert and a subsequent remove strategy (or vice versa) of the same routine symbol.

The highlighted path in the example has a total weight of 1, since the two strategies, insert and remove, can be merged to a logical move strategy having the weight 1. Thus, this repair proposal has the minimal weight and will be ranked higher than other repair proposals. Figure 7.11 shows all repair proposals of the search tree with their respective total weights.

These ordered repair proposals are then used in a wizard as shown in Figure 7.12(a). The end user can then select the adequate repair proposal and the tool automatically applies the changes (Figure 7.12(b)).

These repair proposals are assumed to be valid repair proposals, as stated above. Nevertheless, this assumption might be wrong, if the program repair proposal introduces errors which emerge later in the program. In order to only propose repair proposals that are guaranteed to repair the program, we could apply the change proposals of the best repair proposals to a copy of the defective program and then have the program checked. However, this check may again give false positives in case the program had multiple violations. Program repair only repairs the first violation within a program, since later violations might be consecutive faults. A repair proposal which corrects the first violation does not necessarily make the whole program correct, but eliminates this first violation.

Error Location Before Error Detection

In the example shown above, it was simple to find a repair proposal, because the location where the contract violation was detected was the exact location where a (short) repair proposal could be found. Nevertheless, there might be situations where an error can be fixed by changing the program several statements before the error location. The algorithm is therefore also executed at states prior to the error location, thus creating additional search trees.

To account for the goal of having changes as close as possible to the error location, repair proposals resulting from such an additional search tree farther from the location of error detection have an additional weight.

Program Repair Example with `WAIT` Statement

Listing 7.13 shows a routine of a program consisting of a cooler and a driller component, which are used in parallel. The parallel threads are coordinated by a `WAIT` statement which waits for the cooler to be started, before the driller is started. Note, that the second parallel block starts in line 8. However, the cooler is stopped only after a certain timeout (line 11). Since we cannot be sure that the driller is stopped before the cooler, a constraint is violated.

The program repair algorithm finds that the error location is within a parallel block, thus it allows using the program repair strategy which inserts `WAIT` statements. The repair proposals found are:

- insert `WAIT NOT driller.isStarted()` before `cooler.stop`
- delete call `cooler.stop`

```
PARALLEL
  WAIT cooler.isCooling();
  driller.start();
  driller.down();
  driller.up();
  driller.stop();
||
  WAIT nextItem();
  cooler.start();
  MSG "drilling hole into item";
  WAIT TIMEOUT(3000);
  cooler.stop();
END
```

Figure 7.13: MONACO code with semantic error. The cooler might be stopped, before the driller.

7.3 Program State Visualization

Program state visualization aims at helping program understanding for end users who have to maintain or adapt existing programs. Currently, end users only have two possibilities to get an understanding of an existing program:

- read the source code and try to understand it, and
- run the program to find out what the results are.

These possibilities are not adequate for end users. On one hand, end users do not have the software engineering expertise to be able to understand complex programs in detail. On the other hand, in the automation domain it can be fatal to run a program without knowing the results beforehand.

7.3.1 Overview

To tackle these issues, a program visualization tool has been created, which allows a *design-time* visualization of MONACO programs [Str09]. It visualizes the machine states corresponding to the different positions in a MONACO program *without* executing the program.

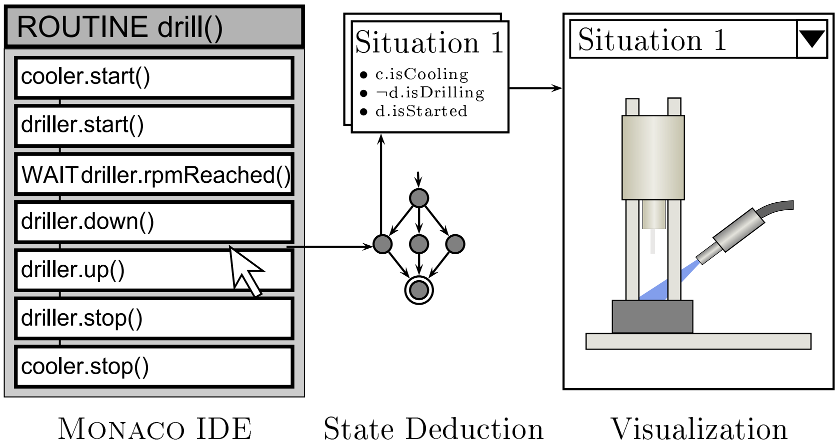


Figure 7.14: Program visualization overview.

The program visualization tool uses situational knowledge created by the state mapping algorithm. The overall process works as follows (see Figure 7.14):

1. The user selects a position in the visual editor of the MONACO IDE without executing the program.
2. The states in the implementation automaton corresponding to the selected statement are searched.
3. The situational knowledge at these states are summed up and forwarded to the visualization tool.
4. The visualization tool uses the situational knowledge to visualize the machine state.

7.3.2 Knowledge Deduction

The knowledge deduction system generates situational knowledge prepared for the visualization tool. The visualization tool gives a list of questions to the knowledge deduction system which in turn computes the answers. The questions are function symbols for which the tool needs the value in order to visualize the component state. The answer to each question can either be *TRUE*, *FALSE*, or *UNKNOWN*, depending on whether the knowledge in a

Function Symbol	Value
$c.isCooling$	$TRUE$
$d.isDrilling$	$FALSE$
$d.isStarted$	$TRUE$
$d.rpmReached$	$UNKNOWN$

Figure 7.15: Results of the state deduction process.

certain situation implies the question ($TRUE$), implies the negation of the question ($FALSE$) or neither of them ($UNKNOWN$).

Assume that we have a cooler and a driller component as shown in Figure 7.14. The user clicks the space after the statement `driller.down()` to see the state of the machine after this statement has been executed. The system finds a single state in the implementation automaton with one situation attached. The knowledge in this single situation is $K = \{c.isCooling, d.isStarted, \neg d.isDrilling\}$.

The visualization asks for the values of all function symbols (in the example $c.isCooling$, $d.isStarted$, $d.isDrilling$, and $d.rpmReached$). From the knowledge above and the invariants of the system, the values shown in Figure 7.15 are deduced using an SMT solver. For each question, the SMT solver needs to verify whether $sat(K \wedge Inv \wedge question)$ or $sat(K \wedge Inv \wedge \neg question)$. If both satisfiability checks are true, or both checks are false, the value $UNKNOWN$ is used. If the location selected by the end user corresponds to several states in the implementation automaton, and/or multiple situations exist, the process described above is executed for each situation. The visualization tool is then provided with the answers for each situation.

7.3.3 Visualization

The state visualization tool is a plugin to the Monaco IDE and displays a schematic view of a set of Monaco components. Based on the values generated from the state deduction (see previous section), the visualization displays parts of components in different colors, size, position, rotation, and visibility and can even run animations. The visualization allows users to switch between multiple situations, so that the end user can see in which states the system could be, when the selected location in the code is reached.

The visualizations of the components, the binding of properties of the components on values of function symbols, as well as the animations need to be created in advance by an expert designing the MONACO component.

Figure 7.16 shows the visualization of a hydraulic solvent can component consisting of a set of valves and a solvent container (from the EcoChargePD case study, see Chapter 8). The visualization shows a picture of the current situtation for the selected position in the MONACO routine. It shows an animation of the solvent flow (arrows in the pipe), the change of the state of a valve (spinning valve symbol), the container filling up with solvent and open (green) and closed (red) valves as well as valves whose states are unknown (gray).

Note, that for the selected code position, the state deduction algorithm has found two different situations (situation 1 is shown currently). The user can switch between the two situations with the arrow buttons in the upper left corner to see the visualization of the state of the components in another situation. Additionally, all function symbols and the values reported by the state deduction process are shown for the selected situation (top right).

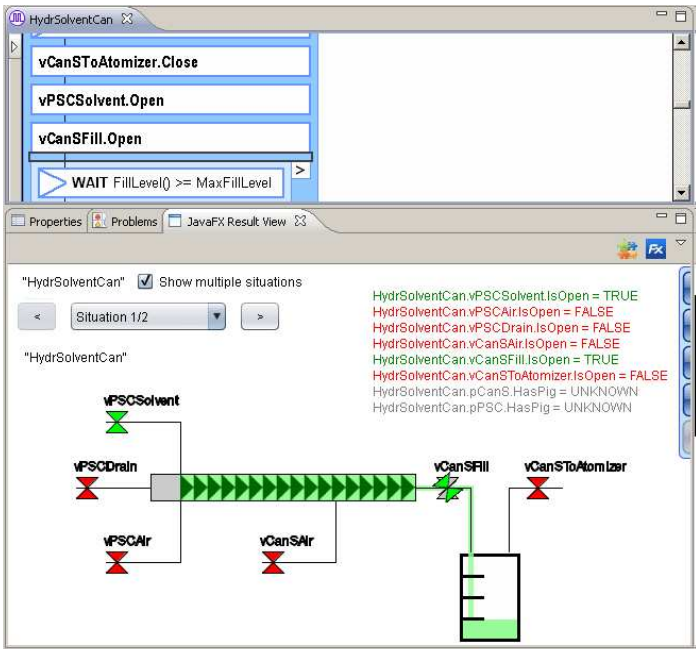


Figure 7.16: State visualization with flow animation.

Chapter 8

Case Studies and Evaluation

This chapter describes case studies in which the presented work has been validated. Furthermore, evaluation results about program state visualization are presented.

8.1 Keplast Injection Molding Machine

The injection molding machine software investigated in this section is a reimplementation of an existing control program of our industrial partner Keba. As the system has already been introduced in Section 3.5 we will refer to Section 3.5 for details on the Monaco implementation.

Recall, that the program is structured into a hierarchy of components (see Figure 3.11). Each component has an interface which defines how it can be used by its upper component.

8.1.1 Contracts

We have created contracts for all interfaces of the Keplast system. The contracts describe the intended usage of the components. We will take a look at the interfaces `IMoldCtrl` and `INozzleCtrl` and their contracts. The interface definition of `IMoldCtrl` is shown in Figure 8.1.

The contract for this interface (see Figure 8.2) allows us to call the `open`

```
INTERFACE IMoldCtrl
  EVENTS error;
  FUNCTION isOpen() : BOOL;
  FUNCTION isClosed() : BOOL;
  FUNCTION clampPos() : REAL;
  ROUTINE open();
  ROUTINE close();
  ROUTINE stop();
END IMoldCtrl
```

Figure 8.1: Interface `IMoldCtrl`.

and `close` routines in turn. It also allows us to call the routine `stop` on the mold, if the routines `close` or `open` are interrupted (by an error signal). The call of the routine `open` has a postcondition that guarantees that after the call the proposition *isOpen* holds. Similarly, the routine call `close` has the postcondition *isClosed*. In addition, the contract also has an invariant, stating that the mold can never be opened and closed at the same time (see Figure 8.3). When an error has interrupted execution of the routines `close` or `open`, the knowledge about the state of the mold is lost (it might be opened, closed, or in an intermediate state). The knowledge about any of these states is therefore retracted (see Figure 8.2).

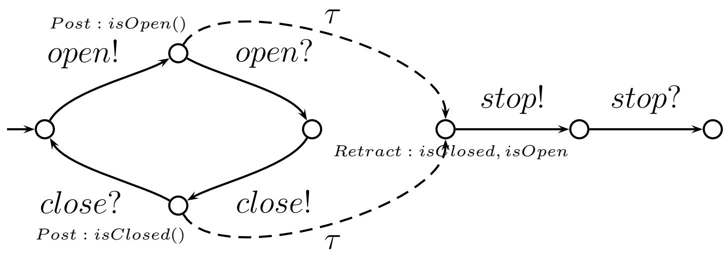


Figure 8.2: Protocol automaton for the interface `IMoldCtrl`.

The second contract we present for the Keplast case study is the contract of the interface `INozzleCtrl` (see Figure 8.4). The nozzle component

```
Invariant: NOT (isClosed() AND isOpen())
```

Figure 8.3: Invariant of `IMoldCtrl`.

```
INTERFACE INozzleCtrl
  ROUTINE startHeating();
  ROUTINE inject();
  ROUTINE plasticize();
  FUNCTION tempReached() : BOOL;
  FUNCTION isPlasticized() : BOOL;
  FUNCTION isInjected() : BOOL;
END INozzleCtrl
```

Figure 8.4: Interface `INozzleCtrl`.

controls the supply with plastic granulate for the injection of melted plastic into a mold. Therefore it has the routines `inject`, `plasticize`, and `startHeating` and the functions `tempReached`, `isPlasticized`, and `isInjected`. The contract of the nozzle specifies, that first the nozzle needs to be heated before the injection routine and the plastification routine can be executed in turn.

The routine `startHeating` guarantees that after its execution the melting temperature of the material has been reached. The routine `inject` needs the nozzle to be filled with plasticized material or to have the target temperature reached and guarantees that after it is executed, the plastic material is injected. Similarly, the routine plasticize guarantees that after its execution the function `isPlasticized` returns true.

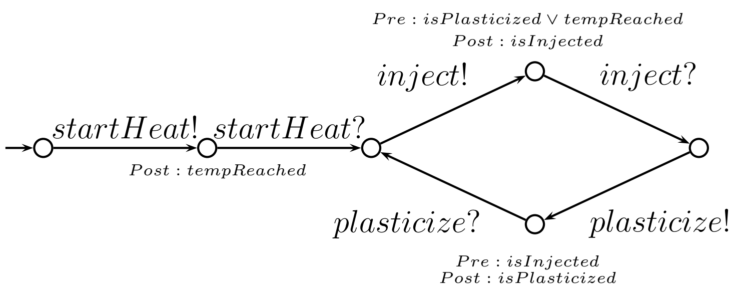


Figure 8.5: Protocol automaton for the interface `INozzleCtrl`.

The contract for `INozzleCtrl` also has an invariant (see Figure 8.6). The invariant states, that the material in the nozzle cannot be refilled (plasticized) and injected at the same time.

```
Invariant: NOT (isPlasticized() AND isInjected())
```

Figure 8.6: Invariant of `INozzleCtrl`.

8.1.2 Constraints

In addition to the contracts, we also identified constraints which need to hold at any time during execution of the system. Figure 8.7 shows a constraint stating that the screw may only be in front, if the heating control has reached the required temperature.

```
CONSTRAINT (IScrewCtrl screw, IHeatingCtrl heating)
    [NOT (screw.isInFront() AND NOT heating.tempReached())]
```

Figure 8.7: Constraint of `IScrewCtrl` and `IHeatingCtrl`.

8.1.3 End-User Support

In this section we will show the application of the different semantic assistance tools in the Keplast case study. All figures will show the tools applied in the routine `Kundenfenster` which is the routine in which end users are supposed to make program changes. First, we will show the semantic assist popup in the MONACO textual editor. Figure 8.8 shows the popup between two parallel statements of the routine. The selected component is mold (since `mold.` is already typed in the editor), and the proposed routine is `open`. According to the contract, no other routine may be called at this position.

Figure 8.9 shows the outline highlighting feature of the MONACO IDE. The spot below the routine call `nozzle.inject` is selected (see mouse cursor) and the outline view shows routines which may be called at this position in the code. The icon of routines that may not be called at this location is crossed out.

Figures 8.10 and 8.11 show the drag-and-drop assistance in the visual editor of the MONACO IDE. Figure 8.10 shows the user dragging the routine call `mold.open` from the outline view to a position where inserting the call is allowed. The immediate feedback of the system is the green plus sign,

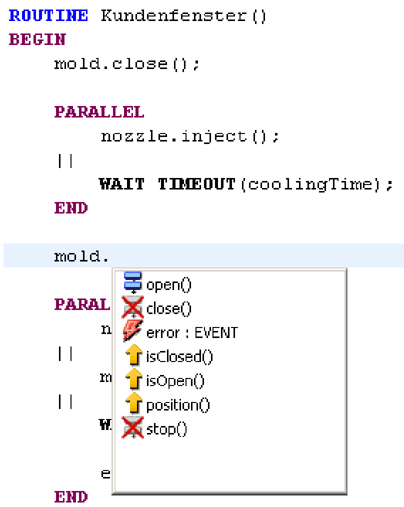


Figure 8.8: Semantic assist popup in the routine `Kundenfenster` proposing routine `open`.

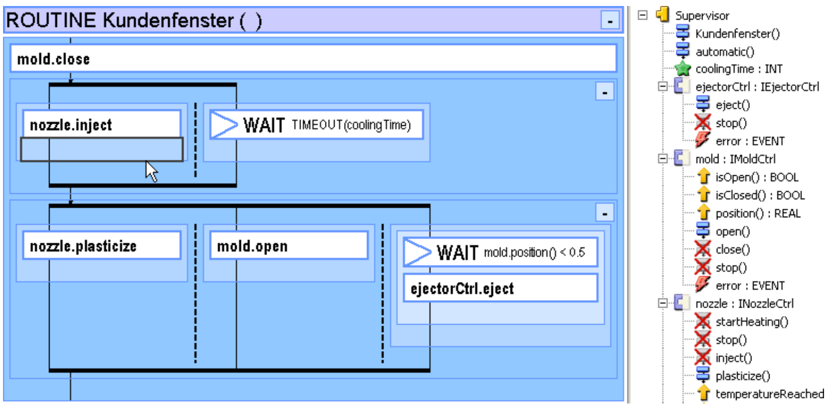


Figure 8.9: Outline highlighting in the routine `Kundenfenster` for the selected position in the code.

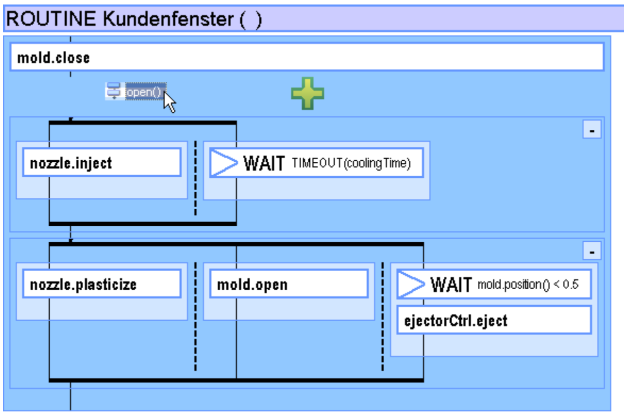


Figure 8.10: Drag-and-drop assistance allowing to insert a routine call.

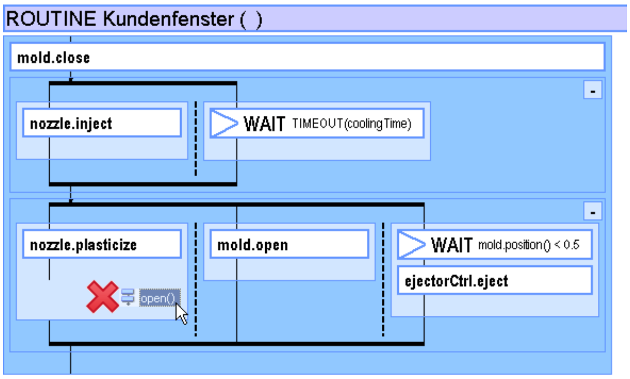


Figure 8.11: Drag-and-drop assistance denying to insert a routine call.

showing that adding the routine call does not lead to a contract violation at that location in the code.

Figure 8.11, in contrast, shows the user dragging the same routine call to a position where it is not allowed to insert the call. A red cross sign indicates that the routine call is not valid here.

Figure 8.12 shows the semantic error resulting from inserting the routine call `mold.open` at an invalid position. In this example the call now

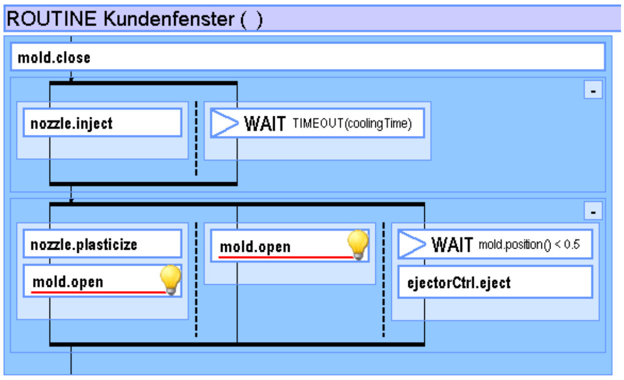
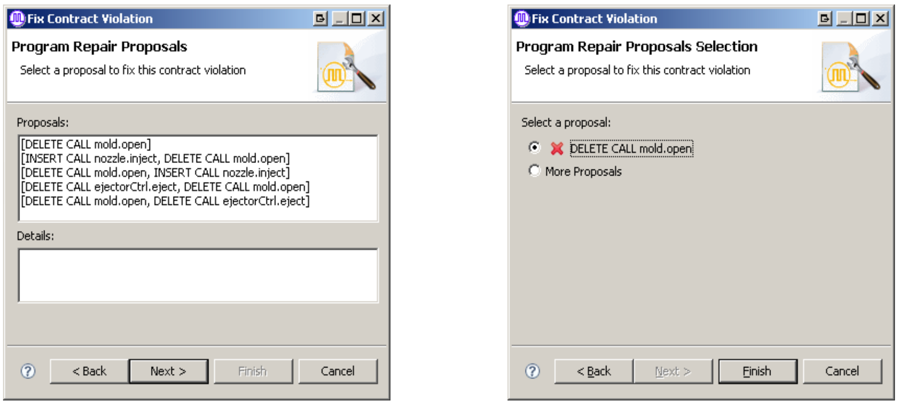


Figure 8.12: Semantic error: the routine `mold.open` is called twice within the parallel statement.

appears in two parallel branches, which is not allowed by the contract. The semantic error is highlighted by a red line and a light bulb. In the example, both instances of `mold.open` are highlighted as errors, since the verification algorithm cannot deduce, which of the calls is an actual error. Clicking the light bulb opens the program repair assistant shown in Figure 8.13. Program repair proposes to remove a call to `mold.open`.

8.2 Duerr Paint Supply System

Duerr is a customer of our project partner Keba and produces painting robots for the automotive industry. We implemented a case study modeling the paint supply system of a painting robot used in the automotive industry (product name: *EcoCharge PD*). The goal of the case study was to show the applicability of MONACO and its tools, including Semantic Assistance, to a system composed of dozens of components. We have reimplemented the reactive control part of the system and proved the applicability of MONACO. In this section, we will describe the system, the contracts of its components, and the constraints we identified. Finally, the application of the various Semantic Assistance tools is shown.



(a) List of proposals. (b) Highest ranked proposal.

Figure 8.13: Program repair proposing to delete one of the calls to `mold.open`.

8.2.1 Monaco Application

The paint supply system consists of six Monaco components and over 60 native subcomponents. It regulates the paint supply and the purging of the paint pipes. The native subcomponents are mostly valves being opened and closed to let paint, air and solvent flow through pipes, and to fill paint pistons. Some of the pipes contain so-called *pigs* (*p*ipeline *i*nspection *g*auge*s*) that float in the pipe and physically separate different liquids or air being transported.

Figure 8.14 shows the main components of the system. On the bottom left, the *color changer* component allows the system to insert different types of paint, without mixing any two colors. Next, a pipe with a pig leads to one of the *subchannels*. The paint supply system may consist of multiple subchannels which independently supply the *atomizer* component (top right) with the exact color needed. The implemented system has two subchannels. While one of the subchannels pushes paint to the atomizer, the other subchannel is reloaded with the appropriate paint for the next product. The atomizer is the spray nozzle that coats the product with the paint. In addition, the *solvent can* component provides the system with solvent for purging the pipes

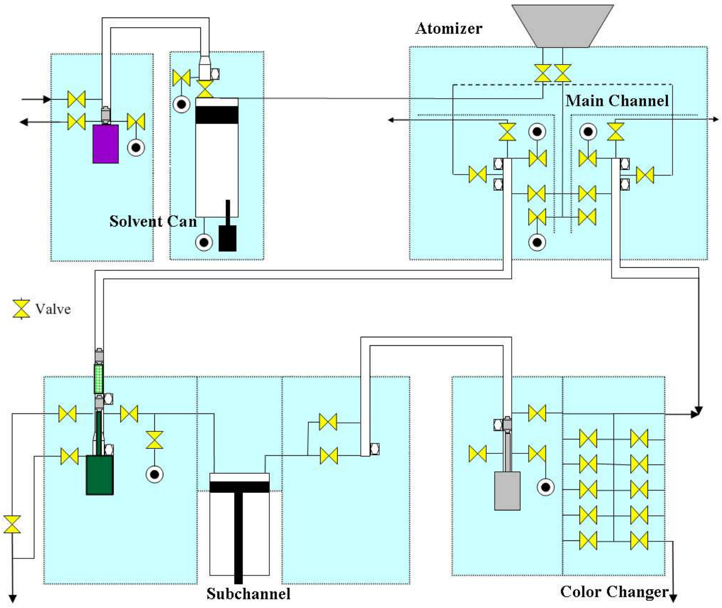


Figure 8.14: Schema of the Duerr application.

whenever a pipe needs to be loaded with another paint.

8.2.2 Contracts

We have created contracts for all interfaces of the Duerr system. The interface `IValve` is used most often, therefore we will discuss this interface and its contract. The interface definition is shown in Figure 8.15. It consists of the function `IsOpen` returning the current state of the valve. In addition, two atomic routines exist, which can be used to open and close the valve.

The contract for this interface is depicted in Figure 8.16. The contract allows opening and closing the valve in turn. Postconditions guarantee that after calling the routine `open` the proposition *isOpen*() holds. Similarly, calling the routine `close` guarantees that the valve is not opened.

Another component in the Duerr application is the solvent can. The sol-

```
INTERFACE IValve
  FUNCTION IsOpen() : BOOL;
  ATOMIC ROUTINE Open();
  ATOMIC ROUTINE Close();
END IValve
```

Figure 8.15: Interface `IValve`.

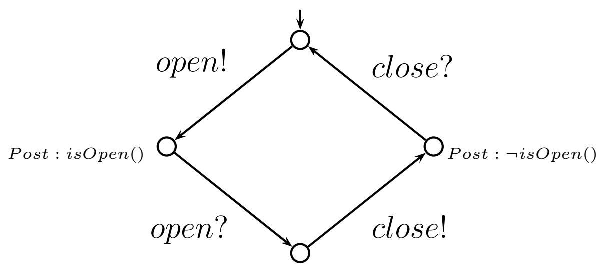


Figure 8.16: Protocol automaton for the contract of `IValve`.

```
INTERFACE ISolventCan
  FUNCTION FillLevel() : INT;
  ROUTINE Init();
  ROUTINE Refill();
END ISolventCan
```

Figure 8.17: Interface `ISolventCan`.

vent can stores solvent to purge pipes which are used to direct different liquids (paint in different colors) in the paint supply system. The can is refilled regularly from a larger solvent tank, and the pipe between this solvent tank and the solvent can needs to be filled with air afterward in order to electrically insulate the tank from the rest of the paint supply system. The interface of the solvent can is `ISolventCan` and is shown in Figure 8.17. It contains the function `FillLevel` and two routines for the initialization (routine `Init`) of the valves and for refilling the solvent can (routine `Refill`).

The protocol automaton for the contract of `ISolventCan` is shown in Figure 8.18. It requires to first call the `Init` routine to initialize the solvent can. Afterwards, the routine `Refill` can be called repeatedly. The contract does not give any guarantees about the component state and does not use preconditions.

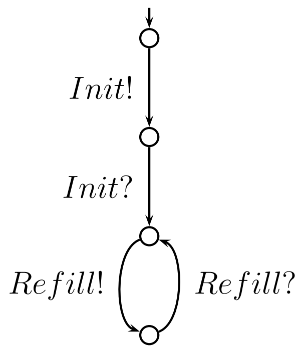


Figure 8.18: Protocol automaton for the contract of `ISolventCan`.

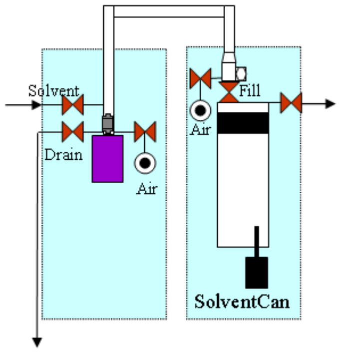
8.2.3 Constraints

We have identified many exclusion conditions that state that certain valves may not be open simultaneously, and modeled these conditions as constraints. In the following, we will take a look at the solvent can component.

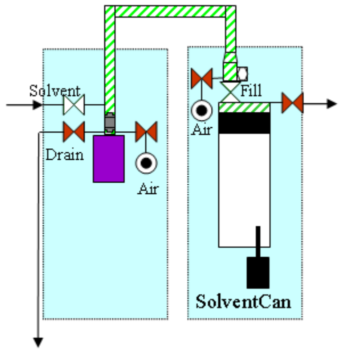
Figure 8.19 shows the solvent can with its valves and pipes in different states, while the solvent can is refilled. The left part of the system is connected to the solvent tank by a valve that brings the solvent to the solvent can. The solvent can (on the right side) is connected to the left part of the system by a pipe. Within the pipe a pig separates solvent from air, such that solvent can be pressed into the solvent can without getting air into the can.

Figure 8.20 shows the exclusions on the valves, meaning that two valves that are connected by a thick red line may never be open at the same time. The exclusions are quite obvious: an air input must never be opened together with a solvent valve, such that no air bubbles are in the solvent. Similarly, the solvent must not be pushed to the drain. The constraints for these exclusions are given in Listing 8.1, for a comprehensive list of all constraints see Appendix B.

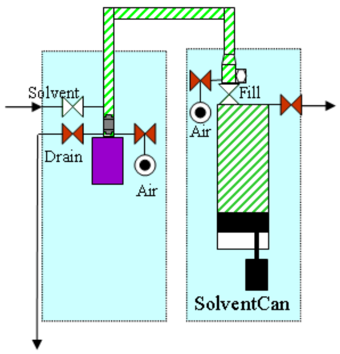
In the original system, these conditions had to be checked at runtime (in every cycle of the execution) and therefore had a great negative impact on runtime resources. These conditions can now be verified statically, even when end users change the program.



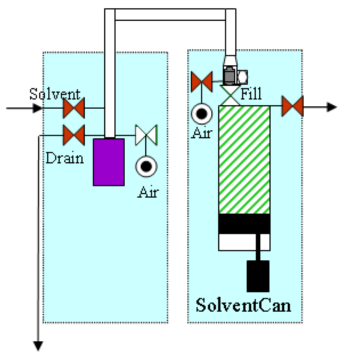
(a) Initial state of the system. All valves are closed.



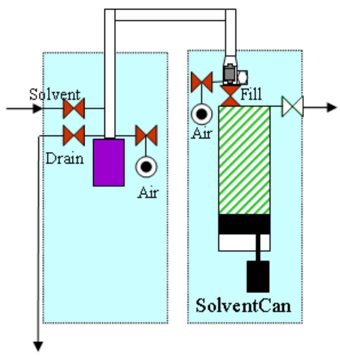
(b) The solvent can is being filled.



(c) The solvent can is getting full.



(d) The remaining solvent in the pipe is pushed into the can using the pig.



(e) When the solvent can is filled, the filling valve is closed and the solvent can be used to purge pipes.

Figure 8.19: Structure and functioning of the solvent can component in the Duerr case study.

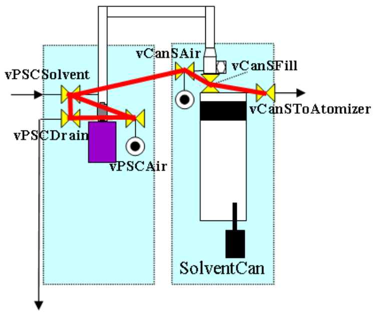


Figure 8.20: Exclusion conditions between the valves of the solvent can component.

```
CONSTRAINT (IValve vPSCAir, IValve vPSCSolvent)
  [NOT (vPSCAir.IsOpen() AND vPSCSolvent.IsOpen())]
CONSTRAINT (IValve vPSCAir, IValve vPSCDrain)
  [NOT (vPSCAir.IsOpen() AND vPSCDrain.IsOpen())]
CONSTRAINT (IValve vPSCSolvent, IValve vPSCDrain)
  [NOT (vPSCSolvent.IsOpen() AND vPSCDrain.IsOpen())]
CONSTRAINT (IValve vCanSAir, IValve vCanSFill)
  [NOT (vCanSAir.IsOpen() AND vCanSFill.IsOpen())]
CONSTRAINT (IValve vCanSAir, IValve vPSCSolvent)
  [NOT (vCanSAir.IsOpen() AND vPSCSolvent.IsOpen())]
CONSTRAINT (IValve vCanSFill, IValve vCanSToAtomizer)
  [NOT (vCanSFill.IsOpen() AND vCanSToAtomizer.IsOpen())]
```

Listing 8.1: Constraints used in the component `HydrSolventCan` case study Duerr.

8.2.4 End-User Support

The different semantic assist tools have also been evaluated in the Duerr case study. All figures will show the tools applied in the routine `Fill` of the solvent can implementation `HydrSolventCan`.

First, Figure 8.21 shows the semantic assist popup in the MONACO text editor after the call to the routine `Open` of subcomponent `vCanSFill`. The selected component is `vCanSToAtomizer`, the valve that connects the solvent can to the other parts of the paint supply system. The only routine proposed is `Close`, since opening the valve would violate a constraint (see Listing 8.1).

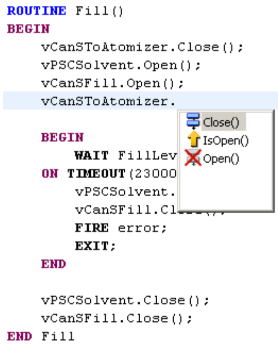


Figure 8.21: Semantic assist popup in the routine `Fill` of component HydrSolventCan.

Outline highlighting is shown in Figure 8.22. The selection is between the routine calls `vPSCSolvent.Open` and `vCanSFill.Open`. In the outline view (right part of the figure) some routines are disabled (icon is crossed out). The routine `Open` of the subcomponent `vPSCAir` is disabled, because a constraint enforces that the valve `vPSCSolvent` and `vPSCAir` are not open at the same time. Directly above the selection, one of the valves is opened, therefore the other valve may not be opened. The routines `vPSCDrain.Open` and `vCanSAir.Open` are invalid for the same reason.

Figures 8.23 and 8.24 show the drag-and-drop assistance in the routine `Fill` of the solvent can component. In the first figure, the insertion of the call is allowed (a green plus sign appears). In the second figure, the call is dragged onto a location where inserting the routine call would violate a constraint. Therefore a red cross sign is shown to indicate this violation.

Figure 8.25 shows the routine `Fill` with a semantic error. The valve `vCanSToAtomizer` is opened although this violates a constraint. This semantic error is highlighted in the visual editor by the red line and a light bulb. The light bulb signalizes that program repair can find a suitable fix for the error.

The program repair results for the semantic error in Figure 8.25 are shown in Figure 8.26. The proposals with the best ranking are to either close the valve `vCanSFill` before the location of the error, or to delete the call causing

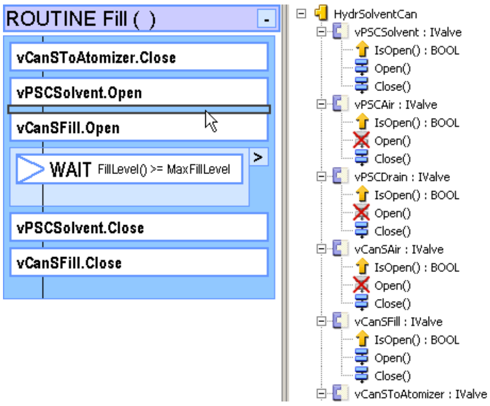


Figure 8.22: Semantic Assistance highlighting valid and invalid routines in the outline.

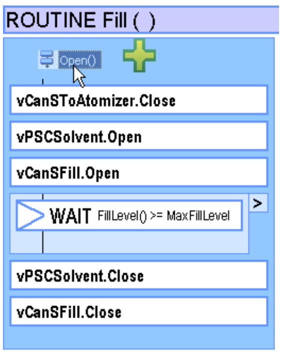


Figure 8.23: Drag-and-drop assistance allows adding the routine call.

the constraint violation.

8.3 Program State Visualization Evaluation

In order to show the effectiveness of program state visualization, an evaluation study with undergraduate mechatronics students was conducted. This study was on end-user programming and its results were — although very

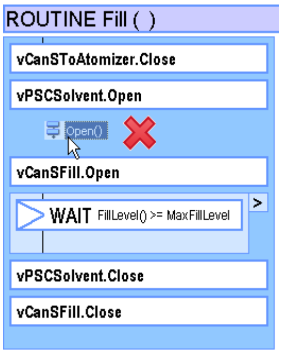


Figure 8.24: Drag-and-drop assistance indicates violation of a contract or a constraint.

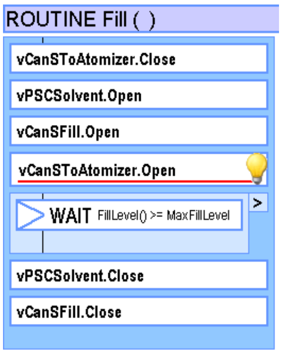


Figure 8.25: Routine `Fill` with a semantic error.

promising — not statistically significant. We therefore are going to set up a second study to probe the benefits of program state visualization on program understanding.

8.3.1 Program Visualization Guiding End-User Programming

The first experiment had the goal to identify the benefit of program visualization for end-user programming. It was conducted with 11 undergraduate

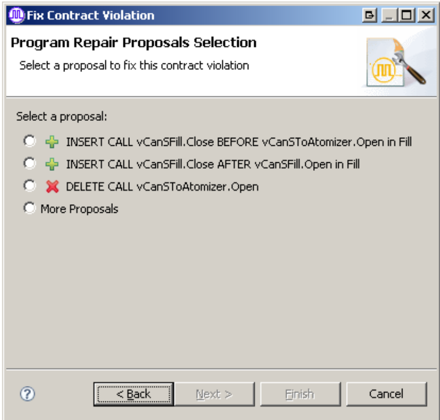


Figure 8.26: Program repair proposals for the semantic error shown in Figure 8.25.

students which were presented a component of a bottle sorting application by means of a video clip of a machine simulation. We introduced the students to the application, MONACO-specific statements, as well as all possible routine calls and conditions. The presentation and introduction was performed in groups of 4 students, such that the students had equal knowledge of the system. The students were then assigned to one of four experiment stations, where they were assigned the task of programming the bottle partition algorithm they had seen before. In order to keep the impact of tool handling and usability as low as possible, an operator trained in using the MONACO system performed the programming tasks as the students commanded.

Each group was (without knowledge of the students) separated into two subgroups, one group being able to use the program visualization, and another group that had to do the programming task without using the program visualization tool. The visualization given to one group of the students is shown in Figure 8.27. It shows the top view of a conveyor belt with two sensors (black dots to the left of the conveyor belt) and two gates which could be used to stop bottles from being moved by the belt. The belt moves bottles from the bottom end to the upper end of the belt, where they are removed by a robot. The task of the students was to create a program that ensures that always at most one bottle was at the removal position (top of the figure).

	Visualization						$\overline{x}$	No Visualization					$\overline{y}$
Skills	3	4	1	1	3	2	2.33	2	2	3	2	1	2.00
Duration [mins]	5	7	2	2	6	3	4.17	5	9	5	3	6	5.60

Table 8.1: Results of the first experiment

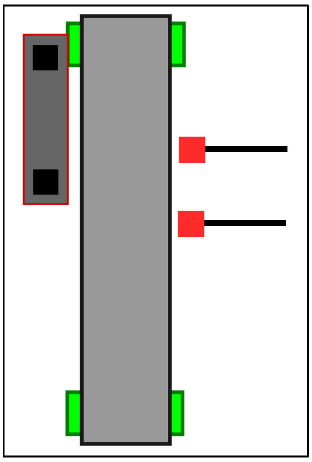


Figure 8.27: Program state visualization used in the first experiment.

The visualization showed the students the current state of the system: whether a certain gate was opened or closed, whether a bottle was at the first sensor (between the gates) or at the second sensor (at the removal position at the end of the belt). The students could use the visualization to think about the next step they wanted the program to perform.

Each student was asked to rate his programming skills on a scale of 1 to 5, with 1 being "very good" and 5 being "poor". This way we could track the influence of general programming skills on the experiment. We measured the time it took the students to implement the program correctly. Table 8.1 shows the results of the individual students in this experiment.

Interpretation

Due to the small sample size, no well-grounded statements can be made. We have seen that program visualization has no significant impact on the productivity of programmers who need to create a program from scratch.

8.3.2 Program Visualization Helping Program Understanding

A second detailed experiment is being planned for the next semester, since the first experiment did not reveal statistically significant data. The experiment will research the benefit of using program state visualization to understand program behavior and find bugs. It will be conducted in the oncoming semester with mechatronics students who will be presented a valve system similar to the paint supply system (see Section 8.2). We will introduce the students to the different components of the application and statements specific to MONACO. Next, the students will have to describe the behavior of a prepared program. We will measure the time it takes the students to fully explain the functionality of the program. As a second test, we will give a similar program to the students, now with a small error introduced. One of the valves is not opened, and thus the fluid can not flow through the system as expected. We will measure again, how long it takes the students to find the error and find a suitable solution to the problem. In both tests, we will also make notes of misunderstandings and false conclusions.

Similar to the first experiment, we will conduct the second experiment with only one half of the students being able to use the program visualization. Both groups will have the MONACO source code of the defective application in the visual editor to find the error. For this test, we will disable highlighting of contract and constraint violations in the visual editor, otherwise finding the error would be trivial. The program visualization for this system is depicted in Figure 8.28.

We have already run this experiment with colleagues as test persons and have seen that the first results are very promising. In order to get statistically relevant data, we will run the experiment with a larger sample size of students.

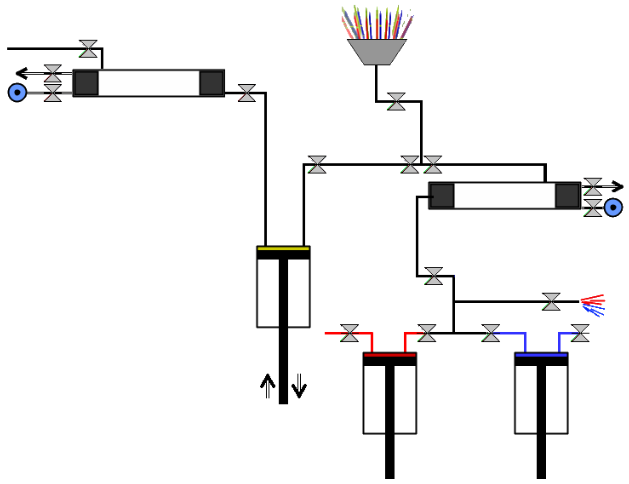


Figure 8.28: Program state visualization that will be used in the second experiment.

Chapter 9

Related Work

This chapter compares different aspects of our work with existing approaches and highlights their differences. Sections 9.1 and 9.2 introduce related work on the verification of call sequences and safety properties. Section 9.3 describes work on automatic repair of programs based on some specification of correctness. Section 9.4 compares work on program visualization to the design-time animation approach.

9.1 Verification of Call Sequences

Verification of call sequence constraints has been investigated by many researchers [OO90, OO92, PV02, HB07, Jin07]. The systems most similar to the work of this thesis are presented in the following.

9.1.1 Cecil/Cesar

Olender and Osterweil describe *Cecil*, a language for the specification of sequencing constraints in a regular expression dialect (AQRE - anchored, quantified, regular expressions) [OO90]. The language can be used to describe valid execution sequences of routine calls of abstract data types. Instead of specifying the complete execution path Cecil expressions describe portions of the valid behavior, therefore allowing partial specification of behavior. Cecil specifications first describe which routine calls they govern. Then a list of partial specifications starting and ending at so-called anchors follows. Anchor routines are written in square brackets, the special anchors `[s]` and `[t]` describe the start and the end of the program, respectively.

Between two anchors, expressions similar to regular expressions can be used to express valid sequences of routine calls. The quantifiers `forall` and `exists` can be used to denote that the following expression needs to be observed in each path of the program execution between the anchors, or in at least one path. The special symbol `?` matches any routine call governed by this Cecil constraint. The operator `*` denotes an arbitrary number of repetitions of the preceding subexpression (including zero times). The operator `+` denotes repetition of the preceding subexpression (at least one time).

Let's look at an example describing call sequences of an abstract data type for writing to files. Reasonable constraints for the available operations (`open`, `close`, `write`) would describe that a file needs to be opened before it can be written and must be closed before a new file can be opened. Furthermore, one could want to ensure that a file is only opened if it is eventually written. Listing 9.1 lists a Cecil constraint for such a file data type.

```
{open, close, write} (
   [s] forall (open; write*; close)* [t]
   and [open] exists ?+ [write] )
```

Listing 9.1: Cecil constraint for a file operation routines

Cesar [OO92] is the constraint checking tool for Cecil expressions. Cesar's sequencing analysis is based on a state propagation algorithm similar to the state mapping algorithm described in Section 6.2. Instead of inlining the flow graph of a callee into the flow graph of the caller, Cesar keeps the flow graph of the callee separate and continues checking of a local routine call in the flow graph of the respective routine. This approach makes it possible to analyze recursive routine calls of abstract data types.

The implementation of Cesar provided tools to analyze Fortran programs, and support for C and Ada programs was announced. In contrast to protocol contracts, Cecil provides no means to specify preconditions, postconditions or invariants to gather information about the abstract data type. In addition, Cecil constraints can not operate on multiple instances of a data type (variables, subcomponents), which is necessary for the component-based approach of MONACO.

9.1.2 Behavior Protocols

Plasil et al. present *Behavior Protocols* [PV02, PJP06, Kof07], a language for the description of component behavior. The language is similar to regular expressions and describes the interaction of components based on the SOFA component model.

SOFA components implement two types of interfaces: *required* and *provided* interfaces. The two types of interfaces can be compared to the component boundaries of MONACO components: subcomponent variables specified by their interfaces constitute the required interfaces, while the interface of the component is the provided interface. The provided interfaces receive events (routine calls in MONACO terms) and the component sends events to the required interfaces. The communication structure of the components in SOFA allows more than MONACOs strictly hierarchical component composition. SOFA allows modeling arbitrary component networks and component interactions. While every MONACO component can only implement one provided interface, SOFA components can have multiple provided interfaces.

Recently, a new approach called *Threaded Behavior Protocols* [KPŠ08], was presented. Threaded behavior protocols separate the provided interface description (*provisions*) from the internal behavior which is again separated into *reactions* and *threads*. Reactions and threads make up the actual behavior of the component, possible spread over multiple threads. In contrast to threaded behavior protocols, our work extracts the actual behavior of a component from the code (implementation automaton), while in threaded behavior protocols the implementation is expected to meet the behavior of the reactions and threads sections of the protocol.

Threaded behavior protocols support three main use cases:

UC1: Correctness Check Given a complete component application, show that it does not contain communication errors.

UC2: Substitutability Given two components, show that one can be replaced by the other in a specific application or in any application.

UC3: Code Conformance Ensure that a component implementation conforms to its behavior specification.

The work presented in this thesis supports all three use cases. The basic use case supported is the code conformance check (*UC3*): components are checked to ensure they conform to their contracts with respect to the contracts of their subcomponents. If all components of an application conform to their respective contracts, the complete component hierarchy is correct (*UC1*).

UC2 is only partly supported by the checking approach presented in Chapter 6: Since our verification approach checks components separately, it is possible to guarantee substitutability of two components, if, and only if, they implement the same interface and thus conform to the same contract.

9.1.3 Interface Grammar

Interface grammar [HB07] is a specification language based on grammars which describe the valid usage of a Java component as a context free grammar. The grammar can be annotated with semantic actions (Java code) and is then used to generate component stubs. These component stubs contain a table-driven top-down parser which regards method invocations as input symbols. A program using these component stubs is then statically checked (using Java Path Finder) to verify that the components are used as specified by their interface grammars. The language and tools are used in a framework for modular software model checking and have been demonstrated on the Enterprise JavaBeans Persistence API.

Figure 9.1 shows the interface grammar for a file component. The grammar describes that a file can be opened and then read or written multiple times. An open file can also be closed. Double angle brackets separate semantic actions from the interface grammar. These actions are generated into the resulting component stubs.

We will take a closer look at the rule `closed`. The rule only accepts the method call `open`, upon which it invokes the open method on some internal file object, returns that it has successfully invoked `open` and applies the rule `opened`. If other any routine is called, while the rule `closed` is active, the second (empty) case statement triggers, which does not report successful execution of any method (no return statement).

An interface grammar compiler generates a Java class for each interface

```
class file implements IFile {
  << File f; ... >>;
  rule start { apply closed; }
  rule closed {
    choose {
    case ?open(): {
      !<< f >>.open();
      return open; apply opened;
    }
    case : { }
    }
  }
  rule opened {
    choose {
    case ?read(): {
      !<< f >>.read();
      return read; apply opened;
    }
    case ?write(): {
      !<< f >>.write();
      return write; apply opened;
    }
    case ?close(): {
      !<< f >>.close();
      return close; apply closed;
    }
    case : { }
    }
  }
}
```

Figure 9.1: Interface grammar description for a file component.

grammar containing a table-driven top-down parser which handles all method calls accepted by the grammar. The resulting Java classes are component stubs, which make sure that the component's routines are called as dictated by their interface grammars. A model checker is then able to statically verify that such a component is used in an orderly manner (the component stubs throw exceptions when an illegal usage is found).

The approach of interface grammar is similar to our approach, in that

```
NOT (vPSCAir.IsOpen() AND vPSCSolvent.IsOpen())
```

Figure 9.2: Constraint for two valves: they should never be open at the same time.

they also aim at finding illegal usage of components by some client code. Their description of component behavior is based on context-free grammars and therefore allows to specify nested method calls. Safety properties, such as the constraints described in this work are not part of the interface grammars.

9.2 Checking Safety Properties

The *SPIN* model checker (Simple Promela Interpreter) [Hol03] developed by Gerard J. Holzmann uses *LTL* (linear temporal logic) [CGP99] to describe *safety* and *liveness* properties. Similar to the notion of constraints, safety properties in LTL assert that nothing bad happens. If we express the constraint in Figure 9.2 in LTL we get $\mathbf{G}\neg(vPSCAir.IsOpen \wedge vPSCSolvent.IsOpen)$. In essence, only the *globally* operator is added. Unlike LTL, the constraints presented in this thesis do not allow stating liveness properties. In SPIN, programs under verification are modeled in *PROMELA* (process meta language) and consist of processes which may communicate with each other.

As most model checking tools, SPIN is also aimed at expert programmers who want to check safety and liveness properties of their code. SPIN provides no support for end-user programmers. SPIN is therefore often used as backend in verification systems, where the program under verification is translated to PROMELA code. Amongst others, behavior protocols (see Section 9.1.2) have been experimentally translated to PROMELA code and then model checked using SPIN [Kof07].

Ball et al. (Microsoft Research) developed a static analysis toolkit called *SLAM* [BR01, BBC+06] that finds API usage errors in C programs. The toolkit is used in the static driver verifier tool (*SDV*) to find kernel API usage errors in Windows device drivers. First, an instrumented version of the code under verification is automatically generated. A tool then abstracts the instrumented code into a so-called Boolean program, consisting of the original control flow constructs and Boolean variables, only. API rules describe the temporal safety properties of the API usage as a state machine. The environment of the device driver (operating system, kernel APIs) is modeled as a C program invoking the device driver and simulating the kernel behavior.

The instrumented and abstracted code together with the environment code is then model checked by a separate tool (BEBOP [BR01]). If a bug is found, the abstraction is refined to find the cause of the bug. This abstraction/refinement loop is continued, until either the bug is confirmed or the bug is found to be spurious.

Microsoft Code Contracts [ABF+09] provide a language-agnostic way to express coding assumptions in .NET programs. The contracts take the form of preconditions, postconditions, and object invariants either stated directly in the code or in so-called interface contracts. The contracts can be statically verified, or checked at runtime. In addition, contracts can be used to generate documentation. Code contracts are similar to the pre- and postconditions and constraints in the contracts described in this work. Their purpose is to help developers of .net applications and libraries to statically verify certain properties of their components, as well as to check the pre- and postconditions at runtime. The purpose of our work, however, is to guide end-users in changing component code based on contracts engineered by professional developers. Out of all tools presented in this section, Microsoft Code Contracts have the best integration into a development environment (Microsoft Visual Studio 2010 beta).

9.3 Program Repair

Jobstmann et al. [JGB05,SJB05,GBHW05] try to fix problems in a program by building a product of the broken program and the specification. They regard this as a *game*, where a winning strategy describes a possible program repair. Program repair is restricted to changes in assignment statements (only changes on the left hand side of assignments), without making changes to the program logic by changing the control flow. Similar to our implementation, they assume a fault localizer (the state mapping algorithm in our system) to find the problems beforehand.

Farn et al. [WC08] define a program repair based on graphical state-transition specifications. They identify four atomic edit operations on the specifications (add and delete states as well as add and delete transitions). The cost of the program repair solely depends on the number of edit operations used. The operations all have equal weight. Our approach, in contrast, uses change operations at a higher level where one operation (e.g., add or remove a routine call) results in several changes to the structure of the model of the program. Moreover, our change operations have different weights, thus favoring certain changes over others.

Error correcting parsers search for changes in an erroneous program to create a syntactically correct program. Röhrich [Röh80] proposes a method by which a stack-based parser is able to recover from a syntactic error in a program by searching for a shortest path of the error state to a terminal state of the parser (*emergency route*). This shortest path is then used to find a match between the next input symbols and the symbols expected on the states of the path to the terminal state. Symbols found in the input denote *anchors*. If an anchor is found, the symbols in the input sequence preceding the anchor are removed from the input, and symbols on the shortest path in the parser's stack automaton are inserted into the input. This approach is similar to our approach in that it tries to adapt the input sequence (implementation automaton in our system) to match the parser's stack automaton (protocol automaton in our system). In distinction to our approach, Röhrich uses an emergency route to a terminal state to find a state where parsing can be resumed.

The problem of program repair is similar to the problem of *approximate string matching* [Nav99]. In approximate string matching, a given string (pattern) is being matched to another string which is equal or similar to the pattern. The metric of closeness (also referred to as *edit distance*) describes the number of mismatching characters in the string, where a mismatch can be corrected by insertion, removal or substitution of a character. The edit distance metric most often used is the *Levenshtein distance* measuring the number of edit operations necessary to change the string such that it exactly matches the pattern.

The relation of approximate string matching and program repair is, that in program repair, the specification forms the pattern which needs to be matched in a program. If the pattern does not exactly match, a mistake was found. The changes necessary to repair the program, are the edit operations. While approximate string matching is able to find matches between a pattern and a string, it is a memoryless strategy which is not able to perform a knowledge update due to edit operations. In addition, the restricted set of edit operations is not sufficient for complex patterns such as contracts with preconditions and postconditions.

9.4 Program Visualization

Techniques similar to program visualization have been used in teaching and debugging algorithms [MS93, BS84]. These systems interact with a running program by either calling the animation part explicitly from the algorithm, or by binding the values of the variables to properties of the animation. Therefore, it is necessary to actually execute (and optionally debug) the animated program. Our system, in contrast, visualizes the states of the components of a program without executing the code, based on the cursor position in the code and state information deduced by our static analysis.

Many other tools for algorithm visualization have been proposed. They mostly aim at helping students learn how to program. These systems can be categorized into two main categories [UFVI09]:

Script-based Systems. In these systems the user needs to manipulate the source code of the program/algorithm being visualized. Calls to the visualization engine are added at certain positions. Executing the program then generates a visualization script, which shows the steps the program has taken (e.g., ANIMAL [RSF00]).

Compiler-based Systems. Compiler-based systems generate algorithm visualizations without changing the source code of the algorithm. The interaction with the visualization system is added to the program automatically by a compiler (e.g., Alice [CDP03]).

We see the program visualization tool developed in this work to be in none of the established categories. In our system, the source code does not need to be changed, in order to create a visualization. Furthermore, the compiler does not adapt the program automatically to interact with the visualization system. The visualization is solely based on the state mapping algorithm and its knowledge update steps. We therefore suggest to introduce a new category for algorithm visualization tools based on *static analysis*.

Chapter 10

Summary and Conclusion

This chapter summarizes our approach on using formal methods to guide end-user programming. It presents the main contributions and recapitulates the main ideas of semantic assistance. Finally, this thesis is concluded with an outlook on future work that would make the semantic assistance tools even more useful.

10.1 Summary

This work presents an approach to support programming in industrial automation by formal verification techniques. The approach allows specifying component contracts and constraints which must be obeyed by client programs and verifies that the client program does not violate them. Based on this verification approach, semantic assistance tools have been implemented to support programmers in writing semantically correct programs. The various semantic assistance tools help programmers to use routine calls in valid sequences, repair programs containing semantic errors, and understand a client program by visualizing the state of the components at a specific location in the code.

We have adopted techniques from formal interface specification [dAH01, Mey86], model checking [CGP99], and knowledge changes [KM91] in this work. Formal interface specification techniques are used to specify sequencing constraints of components, knowledge about state properties of components, as well as inter-component constraints. Model checking and artificial intelligence techniques are then used to verify that a client program obeys the contracts and constraints.

The approach is based on MONACO, a domain-specific language for machine automation programming. It allows programming the reactive part of an automation program and therefore has language constructs to express machine operation sequences, has strong support for dealing with exceptional situations and allows parallel activities. The behavioral model of MONACO is close to StateCharts [Har87], however, an imperative, Pascal-like style of programming is used. Most important, MONACO allows hierarchical abstraction of control functionality by a component-based approach which allows building components with interfaces and hierarchical structuring of components, where upper components are in full control over their subordinates.

Outline of the Approach

Our programming guidance is based on contracts and constraints, which are formal descriptions of the intended behavior of component interfaces (see Figure 10.1). MONACO components and their contracts are translated into automata (1),(2). The state mapping algorithm establishes a mapping between the states in the automaton of a MONACO component and the automata of its subcomponents and may find contract violations (3). In addition, the states of a component are associated with knowledge about the states of its subcomponents. This information is derived from postconditions in the contracts and conditional statements in the component implementation. Finally, the state mapping and associated knowledge is used to verify constraints.

The annotated implementation automaton (4) is then used in various end-user support scenarios. Contract or constraint violations (5) are reported and highlighted at the respective locations in the code editor. Based on the contracts and constraints, the system can propose valid routine calls for a selected location (6). Similarly, a program containing a contract violation can be automatically repaired, based on repair strategies such that the program complies with the contracts and constraints(7). Finally, the system uses the state mapping results at a specific location in the code to visualize the state of the subcomponents at that location.

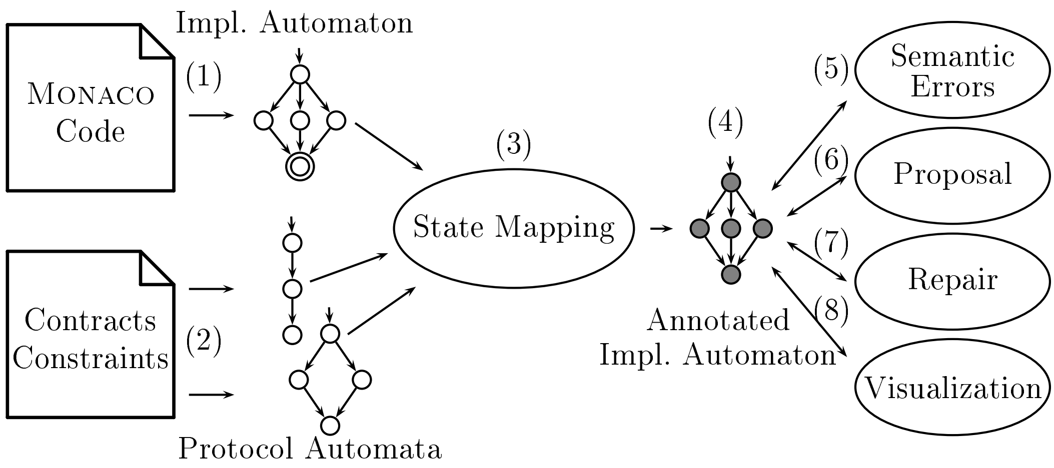


Figure 10.1: Steps in the system for end-user programming guidance.

10.2 Contributions

In the past decade, many verification systems emerged, from general model checkers like SPIN to specific device driver verifiers like SDV. Still, active research is going on in this field to provide tools to verify programs written in general programming languages. To the best of our knowledge, we are the first to base restricted end-user guidance tools on formal methods and verification. The contributions of this work are therefore as follows:

- Contracts allowing to specify the valid call sequences of routines as well as guarantees (postconditions) and required conditions (preconditions).
- Constraints to express safety properties.
- A verification process which checks that a client program obeys contracts and constraints.
- A knowledge deduction process which allows to deduce properties of components fulfilled at particular code positions in the client applications.
- Semantic Assistance tools which propose code fragments based on contracts and constraints and aid in repairing client programs.
- A design-time visualization tool to visualize the state of a system at a position in the code and to help end users understand the program.

10.3 Future Work

Since our system is implemented as a prototype, there are many features that were not implemented but could help the overall approach to be even more effective. This section lists ideas for future work.

- Without changing the overall approach, adding support for routine parameters and local variables could help to get additional information about the possible control flow.
- Although the current notation of contracts is sufficient to describe all possible situations expressible by the automata, a more readable, possibly graphical notation would ease development of contracts. A draft of a better notation is shown in Listing C.2 in Appendix C.
- Postconditions in the contract give guarantees about component states. Such a guarantee holds until it is invalidated by more recent knowledge or it is retracted. Other types of postconditions in a contract would allow the system to guarantee knowledge until the next **`WAIT`** statement, or for a certain period of time only.
- Invariants currently only describe invariant knowledge about a single component. There are situations, in which invariants among several components can be useful to express physical dependencies among different components.
- Similar to systems like WhyLine [KM09], we could extend the knowledge update to preserve the history of the knowledge. We could then not only inform the user which knowledge holds at a certain location, but also give explanations on why particular propositions hold (postcondition, retraction, control flow conditions). Such information would ease diagnosis of semantic errors.

10.4 Conclusions

We feel that there is a natural evolution from the early steps of writing specifications over verification of software systems and debugging to guidance tools and program repair. These tools are valuable not only in the domain of machine automation, but also in other domains where restricted programming by end users is needed, and a similar style of programming is used. The restricted set of features of MONACO eased much of the language-specific parts of the tools. Yet, it seems possible to employ similar tools in more general languages like Java and C$\sharp$, and recent research shows first results [HB07, ABF$^+$09].

Appendix A

Keplast Case Study Constraints

```
CONSTRAINT (IScrewCtrl screw, IHeatingCtrl heating)
    [NOT (screw.isInFront() AND NOT heating.tempReached())]
```

Listing A.1: Constraints used in the case study Keplast.

Appendix B

Duerr Case Study Constraints

```
// Constraints @ HydrSolventCan
CONSTRAINT (IValve vPSCAir, IValve vPSCSolvent)
  [NOT (vPSCAir.IsOpen() AND vPSCSolvent.IsOpen())]
CONSTRAINT (IValve vPSCAir, IValve vPSCDrain)
  [NOT (vPSCAir.IsOpen() AND vPSCDrain.IsOpen())]
CONSTRAINT (IValve vPSCSolvent, IValve vPSCDrain)
  [NOT (vPSCSolvent.IsOpen() AND vPSCDrain.IsOpen())]
CONSTRAINT (IValve vCanSAir, IValve vCanSFill)
  [NOT (vCanSAir.IsOpen() AND vCanSFill.IsOpen())]
CONSTRAINT (IValve vCanSAir, IValve vPSCSolvent)
  [NOT (vCanSAir.IsOpen() AND vPSCSolvent.IsOpen())]

// Constraints @ SubChannel @ HOSE 1
CONSTRAINT (IValve vHose1Drain, IValve vHose1Air)
  [NOT (vHose1Drain.IsOpen() AND vHose1Air.IsOpen())]
CONSTRAINT (IValve vHose1Air, IValve vHose1Color)
  [NOT (vHose1Air.IsOpen() AND vHose1Color.IsOpen())]

// SubChannel @ ATOMIZER
CONSTRAINT (IValve vFMR, IValve vReflowAir)
  [NOT (vFMR.IsOpen() AND vReflowAir.IsOpen())]
CONSTRAINT (IValve vFMR, IValve vMainSolventAVMR)
  [NOT (vFMR.IsOpen() AND vMainSolventAVMR.IsOpen())]

CONSTRAINT (IValve vReflowAir, IValve vRFMRDrain)
```

```
  [NOT (vReflowAir.IsOpen() AND vRFMRDrain.IsOpen())]
CONSTRAINT (IValve vMainSolventAVMR, IValve vRFMRDrain)
  [NOT (vMainSolventAVMR.IsOpen()
      AND vRFMRDrain.IsOpen())]

// Constraints @ MainChannel
CONSTRAINT (IValve vSolvent, IValve vColor)
  [NOT (vSolvent.IsOpen() AND vColor.IsOpen())]

// Constraints @ ColorChanger
CONSTRAINT
  (IValve vColGrey, IValve vColBlack, IValve vColRed,
  IValve vColBlue, IValve vColGreen, IValve vColBrown,
  IValve vColYellow, IValve vColWhite, IValve vColOrange,
  IValve vColPink)
  [
  (vColGrey.IsOpen() AND (NOT vColBlack.IsOpen()) AND
  (NOT vColBlue.IsOpen()) AND (NOT vColRed.IsOpen()) AND
  (NOT vColGreen.IsOpen()) AND (NOT vColBrown.IsOpen()) AND
  (NOT vColYellow.IsOpen()) AND (NOT vColWhite.IsOpen())
  AND (NOT vColOrange.IsOpen()) AND (NOT vColPink.IsOpen())
  ) OR
  (vColBlack.IsOpen() AND (NOT vColGrey.IsOpen()) AND
  (NOT vColBlue.IsOpen()) AND (NOT vColRed.IsOpen()) AND
  (NOT vColGreen.IsOpen()) AND (NOT vColBrown.IsOpen()) AND
  (NOT vColYellow.IsOpen()) AND (NOT vColWhite.IsOpen())
  AND (NOT vColOrange.IsOpen()) AND (NOT vColPink.IsOpen())
  ) OR
  (vColBlue.IsOpen() AND (NOT vColBlack.IsOpen()) AND
  (NOT vColGrey.IsOpen()) AND (NOT vColRed.IsOpen()) AND
  (NOT vColGreen.IsOpen()) AND (NOT vColBrown.IsOpen()) AND
  (NOT vColYellow.IsOpen()) AND (NOT vColWhite.IsOpen())
  AND (NOT vColOrange.IsOpen()) AND (NOT vColPink.IsOpen())
  ) OR
  (vColRed.IsOpen() AND (NOT vColBlack.IsOpen()) AND
  (NOT vColBlue.IsOpen()) AND (NOT vColGrey.IsOpen()) AND
  (NOT vColGreen.IsOpen()) AND (NOT vColBrown.IsOpen()) AND
  (NOT vColYellow.IsOpen()) AND (NOT vColWhite.IsOpen())
  AND (NOT vColOrange.IsOpen()) AND (NOT vColPink.IsOpen())
```

```
) OR
(vColGreen.IsOpen() AND (NOT vColBlack.IsOpen()) AND
(NOT vColBlue.IsOpen()) AND (NOT vColRed.IsOpen()) AND
(NOT vColGrey.IsOpen()) AND (NOT vColBrown.IsOpen()) AND
(NOT vColYellow.IsOpen()) AND (NOT vColWhite.IsOpen())
AND (NOT vColOrange.IsOpen()) AND (NOT vColPink.IsOpen())
) OR
(vColBrown.IsOpen() AND (NOT vColBlack.IsOpen()) AND
(NOT vColBlue.IsOpen()) AND (NOT vColRed.IsOpen()) AND
(NOT vColGreen.IsOpen()) AND (NOT vColGrey.IsOpen()) AND
(NOT vColYellow.IsOpen()) AND (NOT vColWhite.IsOpen())
AND (NOT vColOrange.IsOpen()) AND (NOT vColPink.IsOpen())
) OR
(vColYellow.IsOpen() AND (NOT vColBlack.IsOpen()) AND
(NOT vColBlue.IsOpen()) AND (NOT vColRed.IsOpen()) AND
(NOT vColGreen.IsOpen()) AND (NOT vColBrown.IsOpen()) AND
(NOT vColGrey.IsOpen()) AND (NOT vColWhite.IsOpen()) AND
(NOT vColOrange.IsOpen()) AND (NOT vColPink.IsOpen())
) OR
(vColWhite.IsOpen() AND (NOT vColBlack.IsOpen()) AND
(NOT vColBlue.IsOpen()) AND (NOT vColRed.IsOpen()) AND
(NOT vColGreen.IsOpen()) AND (NOT vColBrown.IsOpen()) AND
(NOT vColYellow.IsOpen()) AND (NOT vColGrey.IsOpen()) AND
(NOT vColOrange.IsOpen()) AND (NOT vColPink.IsOpen())
) OR
(vColOrange.IsOpen() AND (NOT vColBlack.IsOpen()) AND
(NOT vColBlue.IsOpen()) AND (NOT vColRed.IsOpen()) AND
(NOT vColGreen.IsOpen()) AND (NOT vColBrown.IsOpen()) AND
(NOT vColYellow.IsOpen()) AND (NOT vColWhite.IsOpen())
AND (NOT vColGrey.IsOpen()) AND (NOT vColPink.IsOpen())
) OR
(vColPink.IsOpen() AND (NOT vColBlack.IsOpen()) AND
(NOT vColBlue.IsOpen()) AND (NOT vColRed.IsOpen()) AND
(NOT vColGreen.IsOpen()) AND (NOT vColBrown.IsOpen()) AND
(NOT vColYellow.IsOpen()) AND (NOT vColWhite.IsOpen())
AND (NOT vColOrange.IsOpen()) AND (NOT vColGrey.IsOpen())
) OR
(NOT vColGrey.IsOpen() AND  NOT vColBlack.IsOpen() AND
NOT vColBlue.IsOpen() AND  NOT vColRed.IsOpen() AND
```

```
NOT vColGreen.IsOpen() AND  NOT vColBrown.IsOpen() AND
NOT vColYellow.IsOpen() AND  NOT vColWhite.IsOpen() AND
NOT vColOrange.IsOpen() AND  NOT vColPink.IsOpen())
]
```

Listing B.1: Constraints used in the case study Duerr.

Appendix C

EBNF Protocol Contract Notation

Listing C.1 lists the grammar of the EBNF protocol contract notation.

```
SpecEBNF =
    "EBNF" Identifier "=" SpecBlock "." .

SpecBlock = SpecStmts .

SpecStmts = { SpecStmt } .

SpecStmt =
    (
      RoutineCall
    |
      "(" SpecStmts
        (
          { "|" SpecStmts }
          |
          { "||" SpecStmts }
        )
      ")"
    |
      "[" SpecStmts "]"
    |
      "{" SpecStmts "}"
```

```
    )
    [ "on" EventCondition SpecStmt ]
    .

EventCondition = Identifier .

RoutineCall = Identifier .
```

Listing C.1: EBNF Protocol Contract Notation.

Listing C.2 lists a draft of alternative productions for the grammar of the EBNF protocol contract notation. These alternative productions allow to state invariants, preconditions, and postconditions.

```
SpecEBNF =
    "EBNF" [ "<" "Invariant" ":" Condition  ">" ]
    Identifier "=" SpecBlock "." .

RoutineCall = Identifier
              {
                "<"
                  ("Pre" | "Post" | "Retract")
                  ":" Condition
                ">"
              } .

/* Due to reuse of Monaco condition parser, conditions */
/* are parsed as strings.                              */
Condition = { ANY } .
```

Listing C.2: Draft of alternative `RoutineCall` and `SpecEBNF` productions with conditions.

Appendix D

Detailed Protocol Contract Notation

Listing D.1 lists the grammar of the detailed protocol contract notation. The detailed protocol contract notation allows specifying pre- and postconditions as well as initial and invariant conditions.

```
SpecDetail =
    "Interface" Identifier [ Identifier ]
    { "[" "Invariant" ":" Condition "]" } ":"
    {
      ["final"] ["initial"] Identifier { StateCondition }
      "="
      {
        Identifier "." [ Identifier ] ("!"|"?") Identifier
      }
      "."
    } .

StateCondition = "["
    ("Pre" | "Post" | "Retract") ":" Condition "]" .

/* Due to reuse of Monaco condition parser, conditions */
/* are parsed as strings.                              */
Condition = { ANY } .
```

Listing D.1: Detailed Protocol Contract Notation.

Appendix E

Constraint Notation

Listing E.1 lists the grammar of the constraint notation.

```
Constraint =
    "CONSTRAINT"
    "("
      Identifier Identifier
      { "," Identifier Identifier> }
    ")"
    "[" Condition "]"
    .

/* Due to reuse of Monaco condition parser, conditions */
/* are parsed as strings.                             */
Condition = { ANY } .
```

Listing E.1: Constraint Notation in EBNF.

List of Listings

Bibliography

[ABF+09] M. Andersen, M. Barnett, M. Fähndrich, K. King, B. Grunkemeyer, and F. Logozzo. Code contracts, Microsoft Research. URL, http://research.microsoft.com/projects/contracts/, 2009.

[AGM85] C.E. Alchourron, P. Gärdenfors, and D. Makinson. On the logic of theory change: Partial meet functions for contraction and revision. *Journal of Symbolic Logic*, (50):510–530, 1985.

[AL04] Johna Arthorne and Chris Laffra. *Official Eclipse 3.0 FAQs*. Addison-Wesley, 2004.

[ASM80] Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer. Specification language. In *On the Construction of Programs*, pages 343–410. Cambridge University Press, New York, NY, USA, 1980.

[BBC+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 73–85, New York, NY, USA, 2006. ACM.

[BBL08] Robert Brummayer, Armin Biere, and Florian Lonsing. Btor: Bit-precise modelling of word-level problems for model checking. In *Proc. 1st Intl. Workshop on Bit-Precise Reasoning*, 2008.

[BC85] Gérard Berry and Laurent Cosserat. The Esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 389–448, London, UK, 1985. Springer-Verlag.

[BCC98] Sergey Berezin, Sérgio Campos, and Edmund M. Clarke. Compositional reasoning in model checking, 1998.

[BCF+08] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 299–303, Berlin, Heidelberg, 2008. Springer-Verlag.

[BHJM07] Dirk Beyer, T. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST: Applications to software engineering. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, 2007.

[Bie08] Armin Biere. Lecture notes: model checking, JKU Linz, 2008.

[Bje05] Per Bjesse. What is formal verification? *SIGDA Newsl.*, 35(24):1, 2005.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Alexander Pretschner, and Martin Leucker. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[BLR+04] Mike Barnett, K. Rustan M. Leino, K. Rustan, M. Leino, and Wolfram Schulte. The Spec♯ programming system: An overview. pages 49–69. Springer, 2004.

[BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122. Springer-Verlag New York, Inc., 2001.

[Bri09] Walter Bright. Contract programming, Digital Mars. URL, http://www.digitalmars.com/d/2.0/dbc.html, 2009.

[BS84] Marc H. Brown and Robert Sedgewick. A system for algorithm animation. *SIGGRAPH Comput. Graph.*, 18(3):177–186, 1984.

[CDP03] Stephen Cooper, Wanda Dann, and Randy Pausch. Using animated 3d graphics to prepare novices for cs1. *Computer Science Education Journal*, 13:28–29, 2003.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* M.I.T. Press, 1999.

[CL00] Edmund Clarke and Yuan Lu. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169. Springer, 2000.

[dAH01] L. de Alfaro and T. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[dAH05] L. de Alfaro and T. Henzinger. Interface-based design. In *Engineering Theories of Software-intensive Systems*, NATO Science Series: Mathematics, Physics, and Chemistry 195, pages 83–104. Springer, 2005.

[Das03] Satyaki Das. *Predicate abstraction.* PhD thesis, Stanford University, Stanford, CA, USA, 2003.

[DdM06] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[GBHW05] A. Griesmayer, R. Bloem, M. Hautzendorfer, and F. Wotawa. Formal verification of control software: A case study. In *18th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems.*, pages 783–788, Bari, Italy, 2005. Springer.

[GKOT00] Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors. *Abstract State Machines, Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*. Springer, 2000.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[HB07] Graham Hughes and Tevfik Bultan. Interface grammars for modular software model checking. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 39–49, New York, NY, USA, 2007. ACM.

[HJMS03] Thomas Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. page 624. 2003.

[HMP01] Thomas Henzinger, Marius Minea, and Vinayak Prabhu. Assume-guarantee reasoning. In *HSCC. Volume 2034 of LNCS*, pages 275–290. Springer, 2001.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, September 2003.

[HR99] Andreas Herzig and Omar Rifi. Propositional belief base update and minimal change. *Artificial Intelligence*, pages 107–138, 1999.

[Hur06] D. Hurnaus. Eine domänenspezifische Programmiersprache für Maschinensteuerungen: Entwurf eines Compilers und einer Ausführungsumgebung, Master thesis, June 2006.

[HW08] D. Hurnaus and C. Wirth. Model-based generation of domain-specific program environments. In *Workshop on Generative Technologies, Part of ETAPS 2008*, pages 76–77, Budapest, 2008.

[IEC03] IEC. Norm IEC-61131-3 - programmable controllers - part 3: Programming languages, 2003.

[ISO96a] ISO. Vienna development method – specification language – part 1: Base language, Dec. 1996.

[ISO96b] ISO/IEC. ISO/IEC 14977 Extended BNF, Dec. 1996.

[JGB05] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *17th Conference on Computer Aided Verification (CAV '05)*, pages 226–238. Springer-Verlag, 2005. LNCS 3576.

[Jin07] Ying Jin. Formal verification of protocol properties of sequential Java programs. *Computer Software and Applications Conference, Annual International*, 1:475–482, 2007.

[KM89] H. Katsuno and A. O. Mendelzon. A unified view of propositional knowledge base updates. In *Proc. of the 11th IJCAI*, pages 1413–1419, Detroit, MI, USA, 1989.

[KM91] Hirofumi Katsuno and Alberto O. Mendelzon. On the difference between updating a knowledge base and revising it. pages 387–394. Morgan Kaufmann, 1991.

[KM09] Andrew J. Ko and Brad A. Myers. Finding causes of program output with the Java Whyline. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 1569–1578, New York, NY, USA, 2009. ACM.

[Kof07] Jan Kofron. Checking software component behavior using behavior protocols and SPIN. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1513–1517, New York, NY, USA, 2007. ACM.

[KPŠ08] Jan Kofroň, Tomáš Poch, and Ondřej Šerý. TBP: Code-oriented component behavior specification. In *32nd Annual IEEE Software Engineering Workshop*, pages 0–0, 2008.

[McM92] K.L. McMillan. *Symbolic Model Checking – An approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[Mey86] Bertrand Meyer. Design by contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

[Mey92] Bertrand Meyer. *Eiffel the language.* Prentice Hall, 1992.

[Mil89] R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[MS93] Sougata Mukherjea and John T. Stasko. Applying algorithm animation techniques for program tracing, debugging, and understanding. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 456–465, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[Nav99] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, page 2001, 1999.

[OO90] Kurt M. Olender and Leon J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Trans. Softw. Eng.*, 16(3):268–280, 1990.

[OO92] K. M. Olender and Leon J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Trans. Softw. Eng. Methodol.*, 1(1):21–52, 1992.

[PHM06] H. Prähofer, D. Hurnaus, and H. Mössenböck. Building end-user programming systems based on a domain-specific language. *6th OOPSLA Workshop on Domain-Specific Modeling*, Oct 2006.

[PHS+08a] H. Prähofer, D. Hurnaus, R. Schatz, C. Wirth, and H. Mössenböck. A DSL approach for programming automation systems. In *Proc. of SE2008 – Conference on Software Engineering 2008*, pages 242–256, 2008.

[PHS+08b] H. Prähofer, D. Hurnaus, R. Schatz, C. Wirth, and H. Mössenböck. The language Monaco. Internal Report (to appear), Christian Doppler Laboratory for Automated Software Engineering, Linz, Austria, 12 2008.

[PHS+08c] H. Prähofer, D. Hurnaus, R. Schatz, C. Wirth, and H. Mössenböck. Software support for building end-user programming environments in the automation domain. In *WEUSE '08: Proceedings of the 4th international workshop on End-user software engineering*, pages 76–80, New York, NY, USA, 2008. ACM.

[PHWM07] H. Prähofer, D. Hurnaus, C. Wirth, and H. Mössenböck. The domain-specific language Monaco and its visual interactive programming environment. In *Proceedings of Visual Languages and Human-Centric Computing 2007*. IEEE Computer Society, 2007.

[PJP06] Jan Kofron Pavel Jezek and Frantisek Plasil. Model checking of component behavior specification: A real life experience. *Electronic Notes in Theoretical Computer Science*, 160:197–210, 2006.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.

[Rei01] Raymond Reiter. *Knowledge in action: logical foundations for specifying and implementing dynamical systems.* M.I.T. Press, Cambridge, Mass., 2001.

[Röh80] Johannes Röhrich. Methods for the automatic construction of error correcting parsers. *Acta Informatica*, (13):115–139, 1980.

[RSF00] G. Rößling, M. Schüler, and B. Freisleben. The animal algorithm animation tool. In *5th Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000)*, pages 37–40, 2000.

[Sif01] Joseph Sifakis. Modeling real-time systems — challenges and work directions. *Lecture Notes in Computer Science*, 2211:373ff., 2001.

[SJB05] S. Staber, B. Jobstmann, and R. Bloem. Diagnosis is repair. In *16th International Workshop on Principles of Diagnosis*, pages 169–174, Monterey, California, USA, June 2005. Poster.

[Str09] J. Strassmayr, Gasi. Zustandsvisualisierung von Steuerungssoftware basierend auf statischer Programmanalyse und Verifikation, June 2009.

[UFVI09] Jaime Urquiza-Fuentes and J. Ángel Velázquez-Iturbide. A survey of successful evaluations of program visualization and algorithm animation systems. *Trans. Comput. Educ.*, 9(2):1–21, 2009.

[VH00] Willem Visser and Klaus Havelund. Model checking programs. In *Automated Software Engineering Journal*, pages 3–12. Press, 2000.

[WC08] Farn Wang and Chih-Hong Cheng. Program repair suggestions from graphical state-transition specifications. In *FORTE '08: Proceedings of the 28th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems*, pages 185–200, Berlin, Heidelberg, 2008. Springer-Verlag.

[Win90] Marianne Winslett. *Updating logical databases.* Cambridge University Press, New York, NY, USA, 1990.

Dominik Hurnaus
+43 650 723 6 723
dominik.hurnaus@gmail.com

Weichsberg 22
4160 Schlägl
Austria

Curriculum Vitae

October 2009

Work Experience

Sep. 2009 – ...	• Software Engineer at Catalysts GmbH, Linz
Sep. 2006 – ...	• Software Engineer Freelancer
Aug. 2006 – March 2009	• Research assistant and Ph.D. student at the Christian Doppler Laboratory for Automated Software Engineering, Johannes Kepler University and KEBA AG, Linz
March 2006 – June 2006	• Master student at the Christian Doppler Laboratoy for Automated Software Engineering, Johannes Kepler University and KEBA AG, Linz
Nov. 2005 – Feb. 2006	• Internship as Software Engineer, BMW AG, Data bus analysis tools, Dingolfing, Germany
July 2005 – Oct. 2005	• Internship as Software Engineer in Testing, Office Server, Microsoft Corporation, Redmond, WA, USA
July 2003 – March 2005	• Application Developer, VÖEST Alpine Industrieanlagenbau, Linz (part-time from October 2003)
March 2004 – July 2004	• Freelancer: development of a planning tool for hydraulic facilities for VÖEST Alpine Industrieanlagenbau (division continuous casting), Linz
Feb. 2002 – Feb. 2003	• Course instructor 'Fachakademie für angewandte Informatik 4. Semester' (Visual Basic, C#), WIFI Linz
July 2001 – Dec. 2001	• Network technician (international experience: Russia, Brasil) VÖEST Alpine Industrieanlagenbau, Linz

Education

2006 – 2009	• Ph.D. studies, Johannes Kepler University, Linz
2002 – 2006	• Software Engineering, University of Applied Sciences, Hagenberg, summa cum laude
1996 – 2001	• Higher-level secondary commercial college, Rohrbach, summa cum laude
1992 – 1996	• Lower grade highschool, Rohrbach