

Fachgebiete Programmierung eingebetteter Systeme und Softwaretechnik
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin

A Mechanized Verification Environment for Real-Time Process Algebras and Low-Level Programming Languages

vorgelegt von
Diplom-Informatiker
Björn Bartels

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr.-Ing. Uwe Nestmann	Technische Universität Berlin
Berichtende:	Prof. Dr. Sabine Glesner	Technische Universität Berlin
Berichtender:	Prof. Dr. Ernst-Rüdiger Olderog	Universität Oldenburg
Berichtender:	Prof. Dr.-Ing. Stefan Jähnichen	Technische Universität Berlin

Tag der wissenschaftlichen Aussprache: 10. April 2014

Berlin 2014
D 83

Abstract

Nowadays, embedded and reactive real-time systems are often also distributed and operate in dynamically changing environments. Furthermore, these systems handle safety-critical tasks and therefore have to satisfy critical functional and non-functional requirements like, for example, real-time requirements. During development, such systems are often modeled on different levels of abstraction using different formalisms or languages in order to facilitate the verification of crucial properties. Timed process-algebraic formalisms like Timed CSP are well-suited for reasoning about properties of distributed systems on different levels of abstraction. However, it is still a challenging task to establish that implementations given in an unstructured low-level programming language correctly implement the respective process-algebraic specifications. The main challenge is to find a way of overcoming the semantic gap introduced by the different levels of abstraction that ensures that properties established on the higher levels of abstraction are preserved.

In this thesis, we address this problem by developing a mechanized framework that enables the compositional verification of formal relations between timed process-algebraic specifications given in Timed CSP and their implementations given in a low-level programming language. On the level of process-algebraic specifications, we build on an existing mechanization of the operational semantics of Timed CSP in the theorem prover Isabelle/HOL. On the level of programming languages, our framework provides a mechanization and extension of an existing compositional big-step semantics and a Hoare-logic for a basic low-level language. We extend the semantics and the Hoare-logic to support non-determinism and real-time properties and provide mechanized soundness and completeness proofs for the partial and total correctness cases. For proofs about conformance between abstract specifications in Timed CSP and their implementations in the extended low-level language, we use the notion of weak timed bisimulation. As a basis for this relation, we formally derive a labeled transition system from the unlabeled transition system induced by the operational semantics of the extended low-level language. We show how our framework supports the transfer of verification results established using our proof calculus in order to discharge verification conditions resulting from conformance proofs. Furthermore, we provide a CSP-based approach for the specification and verification of distributed reactive systems, which adapt their behavior to changes in their environment. In this modeling approach, the notion of refinement in CSP is exploited to realize a layered specification approach separating the functional behavior of a system from its internal reconfiguration processes.

We have formalized the underlying theory of our approach using the Isabelle/HOL theorem prover. This enables us to mechanically verify all results of this thesis and ensures that critical corner cases are not overlooked. At the same time, we thereby provide a machine-assisted verification environment that enables the mechanized compositional verification of conformance relations between low-level code and timed process-algebraic specifications in dynamically changing environments.

Zusammenfassung

Eingebettete und reaktive Echtzeitsysteme sind heutzutage oft verteilt implementiert und werden in sich dynamisch ändernden Umgebungen eingesetzt. Ihre Nutzung in sicherheitskritischen Umgebungen, erfordert dass diese Systeme neben funktionalen Eigenschaften oft auch nicht-funktionalen Eigenschaften, wie z.B. Echtzeiteigenschaften, genügen müssen. Um besonders wichtige Eigenschaften formal nachweisen zu können, modellieren Entwicklungsprozesse für derartige Systeme diese auf verschiedenen Abstraktionsebenen mit potentiell unterschiedlichen Formalismen und Sprachen. Zeitbehaftete Prozesskalküle beispielsweise ermöglichen den Nachweis kritischer Eigenschaften auf verschiedenen Abstraktionsebenen. Es stellt sich allerdings noch immer problematisch dar, für Implementierungen in einer unstrukturierten Programmiersprache formal nachzuweisen, dass diese sich korrekt hinsichtlich einer abstrakten Spezifikation verhalten und auch deren Eigenschaften bewahren.

In dieser Arbeit präsentieren wir eine mechanisierte Verifikationsumgebung für eingebettete Echtzeitsysteme. Diese ermöglicht die maschinelle und kompositionale Verifikation der Konformität zwischen abstrakten Spezifikationen im Prozesskalkül Timed CSP und Implementierungen in einer maschinennahen und unstrukturierten Sprache. Auf Ebene der prozess-algebraischen Spezifikationen nutzen wir eine bestehende Mechanisierung der operationalen Semantik von Timed CSP im Theorembeweiser Isabelle/HOL. Auf Implementierungsebene bauen wir auf eine kompositionale Semantik und einen zugehörigen Hoare-Kalkül für eine minimalistische unstrukturierte Sprache auf. Diese erweitern wir um Nichtdeterminismus und Echtzeiteigenschaften und stellen maschinengestützte Beweise für Korrektheit und Vollständigkeit sowohl für den Fall der partiellen Korrektheit, als auch für den Fall der totalen Korrektheit bereit. Wir ermöglichen den maschinengestützten Nachweis der Konformität einer Implementierung hinsichtlich ihrer abstrakten Spezifikation basierend auf dem Begriff der schwachen zeitbehafteten Bisimulation. Hierzu leiten wir ausgehend von der operationalen Semantik unserer low-level Sprache formal ein zeitbehaftetes und markiertes Transitionssystem ab. Wir stellen Theoreme bereit, die es ermöglichen, Aussagen unserer Programmlogik in Konformitätsbeweisen zu nutzen. Dadurch erhöhen wir die Wiederverwendbarkeit von Beweisen. Weiterhin präsentieren wir einen Ansatz zur Spezifikation und Verifikation von verteilten und adaptiven Systemen in CSP, welcher es ermöglicht funktionale und adaptive Aspekte eines Systems getrennt voneinander zu spezifizieren und zu analysieren. Die mittels dieses Ansatzes etablierten Eigenschaften lassen sich dann durch unserere Konformitätsrelation von der abstrakten Ebene auf die Implementierungsebene einer unstrukturierten Sprache transferieren.

Die unserem Ansatz zugrunde liegende Theorie formalisieren wir im Theorembeweiser Isabelle/HOL. Dadurch wird die Korrektheit unserer Verifikationsumgebung sichergestellt. Weiterhin erhalten wir eine maschinengestützte Verifikationsumgebung für die kompositionale und mechanisierte Verifikation von Konformitätsrelationen zwischen low-level Programmen und ihren prozessalgebraischen Spezifikationen in sich dynamisch ändernden Umgebungen.

Contents

1 Introduction

When developing safety-critical embedded and reactive real-time software, developers need to achieve a maximum degree of assurance that critical system properties are fulfilled by the executed code. Establishing such properties is a challenging task because the complexity of embedded system software is steadily increasing and embedded applications are approaching the versatility of general purpose software. Besides functional properties, embedded code frequently needs to fulfill non-functional properties like, for example, timing properties. During the development phase of safety-critical systems, functional and non-functional properties of a system are often established on different levels of abstraction, i.e., abstract specifications, high-level implementations and executable code. Often, the properties are closely connected to the limited resources of the actual embedded hardware the software is deployed on. It is therefore important to have development approaches in place that integrate the verification of crucial system properties throughout the different levels of abstraction typically found in embedded software development processes. Furthermore, it is necessary to establish the desired properties in a mathematical concise and machine-assisted fashion in order to gain a maximum of trust into the correctness of the established properties.

1.1 Problem

A variety of specification formalisms and methodologies offer convenient mechanisms for reasoning about the behavior of possibly distributed systems at a high-level of abstraction. These formalisms can be extended to facilitate reasoning about classical functional as well as timing properties. When modeling systems using such formalisms, one usually abstracts from the concrete computation of values or functions. The abstractions keep the resulting models small and ease the process of establishing important properties. However, when implementing systems that are specified using such formalisms, the high level of abstraction introduces a semantic gap between the abstract system models and their final implementations. In such a setting, correctness of an implementation with respect to an abstract specification cannot be trivially established. This is due to the different views on the problem domain taken by the formalisms on the one side and imperative programming languages on the other

side. Taking also transformations from high-level code to low-level representations of a system into consideration, the semantic gap becomes even more evident. Furthermore, non-functional system properties, like timing behavior for example, which can be modeled and reasoned about conveniently on the abstract level, cannot be verified in most general purpose programming languages. However, in many cases conformance between abstract specifications and low-level implementations is of crucial importance. Considering for example safety critical systems, it needs to be assured that certain critical properties not only hold on the abstract level but also for transformations to low-level code. Establishing such problems using formally founded methods is a complex task due to the presence of timing constraints, possible non-determinism and communications. Therefore such proofs require machine-assistance so that critical corner cases cannot be overlooked.

1.2 Objectives

This thesis addresses the problem described above by providing a formally justified verification environment. The environment provides system developers with formal techniques to establish properties about the functional and also the timing behavior of embedded real-time systems on multiple levels of abstraction. We want our framework to fulfill the following criteria :

- **Formal specification of system behavior on different levels of abstraction.** We require the specifications of the developed systems to be precise and unambiguous. The semantics of the employed representations need to be formally defined.
- **Formally justified relation between different levels of abstraction.** To show consistency between different levels of system abstraction, formally justified relations need to be established.
- **Integration of timing behavior on all levels of abstraction.** The proposed approach targets embedded real-time systems. When dealing with such systems, it is important to ensure that certain timing properties are satisfied. We argue that these properties need to be considered on all levels of abstraction, i.e., from the abstract specification down to the low-level representations.
- **Support for adaptivity.** Embedded controllers become more and more versatile and dynamic in the way they control their environment or interact with other systems. Therefore, we require our approach to support adaptive behavior, i.e., systems that change their behaviour in response to changes in their environment.
- **Integration with existing and practically relevant formalisms and programming languages.** To exploit powerful and established formalisms, programming languages, and related tools, we want our approach to support practically relevant languages and formalisms.

- **A high degree of mechanization and automatization.** We want the framework to be mechanized and automated using established proof tools. Mechanization is important in order to guarantee the overall correctness of our approach on the one hand and to guide concrete proofs within our verification framework on the other hand. Mechanization using established proof tools prevents overlooking of corner cases and also facilitates the automatization of proof tasks.
- **Modularity and extendabililty.** We want our verification environment to be modular and extendable. It should serve as a basis for the integration of further formalisms and programming languages.

1.3 Proposed Solution

To achieve the objectives given above, we propose an approach based on the combination of process-algebraic proof mechanisms and interactive program verification [BG10, BG11]. On the abstract specification level, we use process-algebraic formalisms that support reasoning about timing properties. We assume that developers provide an implementation of the abstract specification in a high-level programming language. The high-level code is then transformed to an intermediate low-level language as used in compiler frameworks. We then establish conformance between the implementation given in its intermediate representation and the abstract specification. This has the advantage that we are able to support a variety of programming languages that can be compiled to the chosen intermediate language and avoid formalizing the complex semantics of general purpose languages like, for example, C++.

As the basis for the mechanized and compositional verification of unstructured intermediate code, we formalize a compositional big-step semantics and Hoare-logic for a basic low-level language based on [SU05]. To model external communications, we extend the language to support non-determinism and provide mechanized proofs for the partial and total correctness cases. Furthermore, we extend our simple non-deterministic language to a real-time setting including basic communication primitives and adjust the Hoare-logic appropriately. An adaption of the notion of weak timed bisimulation serves as the foundation for our relation between the low-level implementation levels and the abstract specification level. To be able to employ this relation, we formally derive a timed labeled transition system from the transition system that is defined by the semantics of the aforementioned extended real-time low-level language. By establishing conformance on the basis of timed labeled transition system, we obtain a modular verification environment because we can integrate formalisms and languages that can be interpreted as defining a timed labeled transition system. The compositional specification and verification of concrete bisimulations is supported by the compositional Hoare calculus for our real-time low-level language. We provide abstraction theorems, which enable the transfer of verification results about the compositional big-step semantics to the level of the timed labeled transition system. On the abstract level, we

develop an approach for the specification and verification of adaptive systems. In this approach, we separate the specification and the analysis of adaptive behavior from functional behavior. This leads to a layered specification and verification approach that is especially well suited for process-algebras, which provide a notion of refinement. In combination with our conformance relation, we can ensure that properties established on the level of the abstract specification also hold for low-level implementations. To ensure the overall correctness of our verification environment on the one side and provide mechanized and partly automatized tool support for concrete verification efforts on the other side, we mechanize our verification environment using the theorem prover Isabelle/HOL [NPW02].

To demonstrate the practical relevance of our formal framework, we instantiate our verification environment mentioned above with a concrete process-algebraic formalism. We choose CSP [Hoa85] and its conservative extension to real-time, Timed CSP [Sch92], as process-algebraic formalisms und build on an existing formalization of the operational semantics of Timed CSP [Göt12] in Isabelle/HOL. Choosing CSP and Timed CSP as formalisms enables us to employ established proof tools for the analysis of abstract specifications. Besides the well-known refinement model checker FDR2 [GRA05] for CSP that also provides limited support for Timed CSP, ProB [LF08] offers the possibility to check temporal logical formulas on CSP models and to animate Timed CSP models. For the implementation level, we provide a formalization of a basic low-level language that can be used as a basis for the formalization of concrete low-level representations. In our case study, we instantiate our basic low-level language to a subset of the intermediate representation (IR) of the Low-Level Virtual Machine (LLVM) framework[LA04]. The LLVM IR is a widely used compiler intermediate representation that can be used as the target for the compilation of a variety of high-level programming languages. Our approach can therefore in principal be used with any high-level language that can be compiled to this IR. This further motivates the choice of a low-level representation as the basis for our conformance relation between high-level specifications and their implementations.

By integrating the well-known process algebra Timed CSP and our basic low-level language into a formally founded and mechanized verification approach, we are able to support a broad range of systems on the specification level and provide a basis for the formalization of a variety of implementation languages on the implementation level.

1.4 Motivation

Software engineering for embedded real-time systems is an active field of research and the usage of embedded systems is steadily increasing. Although still being relatively small compared to general purpose software, the complexity of these systems is increasing as well. This is due to the fact that embedded controllers become more versatile. While originally focused only

on specialized tasks, nowadays these systems are dynamically adapting their behavior to changes in the environment. An important factor that diversifies software engineering for embedded systems from general software engineering approaches is that embedded software needs to perform these complex tasks using only limited resources. Another one is given by the potential consequences of malfunctions. While in some areas problems caused by incorrect behavior of embedded controllers might be tolerable, e.g., multi-media systems, malfunctions are clearly not tolerable in the area of safety and security critical systems. When handling critical tasks, unpredicted behavior might cause high financial losses or even endanger human lives. It is therefore necessary that systems that operate in and interact with safety critical environments are designed, developed, and analyzed with greatest care.

Informal design methods and testing in general can only provide limited confidence regarding the correctness of systems (e.g., testing can only show the presence of potential failures) during the different stages of embedded development processes. In contrast to that, formal methods can be used to achieve guarantees (e.g., also show the absence of potential problems) during all stages of development. Formal models of a system design can be used to rigorously analyze a system design on all levels of abstraction and connect the different levels by well founded formal notions. It is thereby possible to avoid design flaws that might otherwise be discovered late during the development cycle. Faults that are, for example, detected in the final implementation by testing might in turn lead to high costs for restructuring and reimplementing the whole system.

The trade-off for the high degree of assurance when using formal design methods is the high amount of manual verification work that might be necessary. The complexity of such proofs, especially in the context of low-level code, introduces the risk of subtle corner cases being overlooked. This makes mechanization and automation especially important. Furthermore, a lot of specialized formal design methods have emerged focusing on different areas of system development. Regarding their usage in embedded design approaches, it is necessary to take a holistic view on verification. By holistic, we mean that connections between specialized methods need to be established. It should be possible to preserve or reuse properties verified using a specialized verification method for a certain level of abstraction on another level of abstraction using possibly another verification method. It is therefore necessary to develop verification approaches that ensure the preservation of established properties throughout different levels of abstraction.

1.5 Main Contributions

The contributions of this thesis are of theoretical as well as of practical relevance to the field of software engineering for safety-critical embedded systems. The main contributions can be summarized as follows:

- Mechanized compositional semantics and proof calculi for low-level languages supporting non-determinism and real-time including mechanized soundness and completeness proofs.
- Transfer of formal conformance relations to process-algebraic specifications and their low-level implementations in the setting of embedded real-time systems.
- An approach for the specification and verification of bisimulation relations based on Hoare specifications.
- A formalization of the semantics of a subset of the LLVM IR and a proof calculus for reasoning about programs given in LLVM IR.
- An approach for specifying and verifying adaptive systems and models of their implementation using CSP.
- A mechanized proof environment for establishing conformance between abstract specifications and their respective low-level representations.

1.6 Context of this Work

The work presented in this thesis was motivated by and carried out within the project VATES[1] [GBGK10, BGG10] funded by the DFG. It is the goal of the project to provide development processes for safety-critical embedded real-time systems with formally founded methods and tools that support the entire development process. The project assumes a top-down approach that starts with an abstract specification, which is further refined and transformed to an implementation. More precisely, the considered development approach starts with an abstract specification given in Timed CSP. To this end, a formalization of the operational semantics of Timed CSP in the theorem prover Isabelle/HOL was developed within the project. This formalization [Göt12] especially enables the mechanical verification of infinite (e.g., parameterized) real-time systems. Moreover, the automatic analysis of finite process descriptions is supported by translations from Timed CSP to the input languages of the model checkers UPPAAL [BY04] and FDR2 [GRA05]. To show that a given implementation correctly implements an abstract specification given in Timed CSP, the implementation is assumed to be transformed to the intermediate representation of the LLVM compiler framework. An algorithm was designed [KH09] and implemented [KBG+11] that automatically extracts a low-level CSP model from LLVM IR code which can then be used to show that the implementation correctly refines the abstract specification. The relation between the different levels of abstraction considered in the VATES project are a main motivation for the work presented in this thesis. The results presented provide the groundwork for the comprehensive verification of the aforementioned extraction algorithm. Furthermore, the formalized concepts yield a verification environment, which is suitable for the verification of

[1] Verification and Transformation of Embedded Systems

low-level implementations with respect to abstract specifications, for example in the context of infinite state systems.

1.7 Outline

This thesis is structured as follows. Chapter 2 introduces formalisms, languages and tools that are used throughout the subsequent chapters. Then in Chapter 3, we proceed by discussing work that is related to the approach presented in this thesis. Afterwards in Chapter 4, we present our verification approach that integrates different levels of system abstraction in more detail. The following Chapter 5 explains the mechanization of our extended operational semantics and related Hoare-style proof calculus for a basic low-level language. The extension to a real-time setting with basic communication facilities is discussed afterwards in Chapter 6. Building on these concepts, we construct a formal relation between low-level languages and abstract specifications given in the process algebra Timed CSP in Chapter 7. In Chapter 8, we explain how distributed adaptive systems can be formalized and analyzed using Timed CSP in order to exploit the concepts from the previous chapters. Case studies that use the presented methodology are presented in Chapter 9. In Chapter 10, we give a conclusion and discuss possible extensions to the concepts presented in this thesis.

2 Background

In this chapter, we give the necessary background information regarding the formalisms and languages used in this thesis. We start with an introduction to the basic Hoare calculus for a simple high-level language. Afterwards, in Section 2.2, we introduce the the Low Level Virtual Machine (LLVM) compiler framework and its intermediate representation (IR) are introduced. The IR serves as an example for a typical low-level language. In Chapter 5, we present a Hoare logic for such languages. The main goal of our framework is to provide methods to show the formal conformance between process-algebraic specifications and their low-level implementations. To realize this relation, we will interpret the semantics of the respective languages as (timed) labeled transition systems. These can be analyzed using several notions of bisimulation. Labeled transition systems and bisimulations are therefore introduced in Section 2.3. After that, the semantics of CSP and its timed extension Timed CSP are presented. While both are capable of specifying and analyzing distributed processes, the latter additionally provides mechanisms for reasoning about timing properties of distributed systems. We mainly focus on the operational semantics here, as the core of our formal framework is based on this semantics. Our framework is mechanized in Higher Order Logic (HOL) using the theorem prover Isabelle. We finish this chapter with a short introduction to HOL and the mechanisms of the interactive prover in Section 2.5 and a brief summary in Section 2.6.

2.1 Hoare Calculus

In this section, we shortly motivate and introduce the axiomatic approach to the specification and verification of properties for structured programming languages. For a detailed introduction we refer to [Rey98] and [AdBO09]. This prominent method for specifying and verifying properties of programs using assertions goes back to Floyd [Flo67b] and Hoare [Hoa69] and is sometimes called the Floyd-Hoare calculus or for short Hoare calculus. The main idea is that the meaning of a program is specified by a description of the state, i.e., the values of some variables, before a program is executed and a description of the state after the program was executed (assuming the program was started in a state which fulfills the aforementioned description of the state). This intuition

of a specification is formalized using the notion of so-called Hoare-triples:

$$\{P\}\ \mathit{Program}\ \{Q\}$$

Here, the *precondition* P and the *postcondition* Q are assertions, which in general are functions from the type of program state to the domain of truth values (the Booleans). The program state describes the values of variables at a given point within a program. Between the two assertions P and Q, the *Program* is given in terms of the formal syntax of the programming language for which the Hoare-calculus is defined. To obtain a proof system that supports the analysis of complex specifications about the behavior of a program given in a certain programming language, the general approach is to first define the behavior of the programming languages basic instructions in isolation and then provide rules that can be used to derive more complex specifications from these basic specifications. The ingredients of such a proof system are called axioms and inference rules. Inference rules have the following format:

$$\frac{\mathit{Premise}_1,\ ...,\ \mathit{Premise}_n}{\mathit{Conclusion}}\ \mathit{Condition}$$

Inference rules can be applied in a logical context where all of the premises $\mathit{Premise}_1$ to $\mathit{Premise}_n$ given as assertions above the line evaluate to true and also a possible side condition *Condition* holds. If this is the case, then one can conclude that the *Conclusion* under the separating line holds as well. Axioms in turn are inference rules that have no preconditions. A proof system and its axioms and inference rules can be analyzed in terms of its correctness and completeness. The operational or denotational semantics of the programming language for which the proof system is defined is used as the basis for such an analysis. If it can be established that the rules of the proof system can only be used to derive specifications about programs which are valid with respect to the operational semantics, then the proof system is called *sound*. If a proof system is capable of providing rules to derive every possible specification about a programming language the system has the property of *completeness*. Note that the notion of *completeness* is often referred to as *relative completeness* [Coo78]. In this context, relative means that the proof system is complete with respect to the logic that the assertions are formalized in.

In relation to the concepts of operational and denotational semantics, the approach to the specification of a meaning of a program taken by Hoare calculi is sometimes called axiomatic semantics. However, in [Rey98] it is argued that a more appropriate name for this approach is specifications and proofs because rather than defining the overall semantics of a program fragment, specifications in this proof system might in general concentrate on more abstract properties of a program fragment.

2.1.1 Axioms and Inference Rules

In the remainder of this section we shortly discuss the main axioms and inference rules for a typical abstract high-level language, often called *while* language.

The first rule is the assignment axiom, which formalizes an instruction which assigns the value a to a program variable x:

$$\{Q[x \mapsto a]\}x\texttt{:=}a\{Q\}$$

The instruction *skip* has no effect on the state and therefore any assertion P that holds before the execution of *skip* also holds after the instruction was executed:

$$\{P\}\texttt{skip}\{P\}$$

The rule for the alternative instruction, which executes either the program fragment s_{true} or s_{false} depending on the evaluation of a Boolean condition b is given as follows:

$$\frac{\{b \wedge P\}s_{true}\{Q\} \quad \{\neg b \wedge P\}s_{false}\{Q\}}{\{P\}\texttt{if } b \texttt{ then } s_{true} \texttt{ else } s_{false}\{Q\}}$$

Here the Boolean condition b is a predicate that depends on the values of the program variables.

The rule for the `while` instruction is considered to be the most demanding proof rule because it requires the specification of an invariant I. The `while` instruction executes a program fragment s_{true} (called the loop body) repeatedly as long as the predicate b evaluates to true before each execution.

$$\frac{\{b \wedge I\}s_{true}\{I\}}{\{I\}\texttt{while } b \texttt{ do } s_{true}\{\neg b \wedge I\}}$$

The invariant I is an assertion which needs to evaluate to true before and after each execution of the loop body but not necessarily during the execution of s_{true}. The intuition is that I relates program variables before and after each execution of the loop body. If the loop condition b does not hold anymore, we therefore know that in the state after the last execution of s_{true} I holds, but b does not.

The rule for sequential composition enables the construction of programs that consist of more than one instruction:

$$\frac{\{P\}s_0\{R\} \quad \{R\}s_1\{Q\}}{\{P\}s_0\texttt{;}s_1\{Q\}}$$

The inference rule uses an intermediate assertion R to compose two program fragments. If after the execution of fragment s_0 that starts in a state fulfilling the assertion P and ends in a state fulfilling R, the fragment s_1 is executed from this state and results in a state that fulfills Q, it can be concluded that

the sequential composition of the two fragments executed in a state fulfilling P ends in a state fulfilling Q.

At the heart of any Hoare-style proof system lies the rule of consequence. Using the previously presented rules, it is only possible to derive specifications that exactly reflect the behavior of the operational semantics of the programming language at hand. The following rule allows us to establish more abstract specifications:

$$\frac{P \models P' \quad \{P'\}s\{Q'\} \quad Q' \models Q}{\{P\}s\{Q\}}$$

The rule of consequence expresses that for any given valid specification, the precondition can be *strengthened* and the postcondition can be *weakened*. The intuition is that the precondition of a valid specification can be replaced by a more restrictive one and it is still ensured that the original postcondition holds after execution of the program fragment. Furthermore, the postcondition can be replaced by a less restrictive assertion.

In Chapter 5, we formalize a Hoare calculus for a low-level language. The inference rules and axioms of that calculus are quite similar to the ones presented in this section. However, as we have to account for unstructured control flow, the assertions will explicitly mention a program counter pointing to the instruction that is to be executed next. In the next section, we briefly introduce a concrete example for such a low-level language.

2.2 Low Level Virtual Machine (LLVM)

In this thesis, we present a mechanized framework that can be used for the verification of conformance relations between low-level programs and their process-algebraic specifications. To this end, we also provide a Hoare-style calculus for the verification of low-level code. The intermediate representation (IR) of the Low Level Virtual Machine (LLVM) compiler framework can be regarded as a typical low-level language in terms of its unstructured control flow. Later in this thesis, we formalize a subset of the language's instructions using our mechanized framework. Furthermore, the development process considered within the project VATES targets intermediate representations as the verification basis for conducting correctness proofs about given implementations. As a concrete representation, we have chosen the LLVM IR. A main advantage of this decision is the versatility of the framework for practical purposes. The modular design of the framework (and also the IR) enables us to cope with a variety of high-level languages and also architectures used in practice. Moreover, the framework is rather self-contained, .i.e., all necessary transformation, optimization, and code generation processes can be realized using the framework. In this section, we give a short overview of the LLVM framework and its intermediate language.

2.2.1 Compiler Infrastructure

The LLVM compiler infrastructure [LA04] provides a modular framework that can easily be extended. There already exist front-ends for various high-level languages that transform programs given in the syntax of the respective high-level language to the internal representation of the LLVM framework, its intermediate language. On this level, so-called compilation passes are applied to the representation of the original program. LLVM offers a rich set of predefined analyses and optimizations that can be used to build further analyses or optimization procedures. For example, the tool LLVM2CSP [KBG+11] developed in the VATES project for extracting CSP models from LLVM code is realized as such a compilation pass. For code generation, back-ends for all commonly used architectures exist.

2.2.2 Intermediate Representation (IR)

The central part of the compiler infrastructure is its platform-independent intermediate representation (IR). The IR is a RISC-like language, which is used internally as the basis for transformations and optimizations. The LLVM IR is a three-address code IR and provides an infinite virtual register set in static single assignment (SSA) form. Furthermore, the IR provides a rather simple language-independent type system, which due to its structure supports the concrete implementations of commonly used high-level features. It can therefore be used as the target for transformations from various high-level languages. This includes weakly typed languages like C for example. It is also possible to define type-safe subsets of LLVM.

Structure of LLVM IR Code In general, the LLVM IR is a typical low-level language as it consists of branching instructions, which allow the control flow to be transferred to potentially arbitrary locations within a given program. However, to impose a basic structure to programs the intermediate language makes use of the concept of basic blocks. Basic blocks are groups of instructions that have a named label. Instructions within a basic block are executed sequentially. The labels of basic blocks are also the targets of branch instructions, i.e., the control flow is not arbitrarily unstructured because jumps are not allowed to arbritrary locations in the program. More structure in LLVM IR programs is possible using function definitions. Within function definitions local variables can be defined and used.

Syntax of the LLVM IR The syntax and semantics of the LLVM IR is defined informally in [LA08]. We summarize the main features about the language here briefly.

The syntax of LLVM is reminiscent of assembler code with type information. For example, the following instruction subtracts the value stored in the

virtual register $\%op\_2$ from the value stored in the virtual register $\%op\_1$ and stores the result in the register $\%dest\_id$:

$...l : \%dest\_id =$ `sub` *type* $\%op\_1, \%op\_2$.

The virtual registers are named by natural numbers prefixed by a %. Before such an instruction can be executed, the operands are loaded from the memory:

$...l : \%op\_1 =$ `load` *type* $\%variable$,`align 4`.

Variables are prefixed by a % as well. Although being a low-level representation, crucial high-level information is explicitly present in the code. This includes information about the data and control flow, as well as type information even for pointer arithmetic. The type system consists of integer and floating-point types as primitives, from which more complex types like arrays and structures are derived. Rich type systems from high-level languages are lowered to the types of the IR. The LLVM IR can express weakly-typed languages like C, as well as defining type-safe subsets of LLVM. In [BG10], we give an operational semantics for a subset of the LLVM IR that we outline briefly in Chapter 9.

In this section, we have introduced the IR of the LLVM framework. The language is a typical example of a low-level language with unstructured control flow. Using our mechanized verification environment, we enable conformance proofs between low-level languages and process-algebraic specifications. To enable such a relation, we interpret the semantics of the low-level language and of the process-algebra as a labeled transition system. In the next section, we therefore give a short introduction to labeled transition systems. Furthermore, we introduce the notion of bisimulations. Bisimulations can be used to relate labeled transition systems with identical behaviors.

2.3 Labeled Transition Systems and Bisimulations

Transition systems (TS) and labeled transition systems (LTS) are important concepts in computer science because a variety of languages and formalisms can be interpreted as defining either a TS or an LTS. The notion of bisimulation allows to identify states within an LTS from which equivalent behaviors are possible. In Chapter 7, we build on these concepts in order to relate process-algebraic specifications to low-level code. Here, we give a short introduction to these formal notions.

2.3.1 Labeled Transition Systems

The basic definition of a transition system (TS) defines an abstract machine, its states, and transitions that are allowed between the states of the abstract machine:

Definition 1 (Transition System)
An TS is a tuple (S, T), where S is a set of states and $T \subseteq (S \times S)$ is a transition relation.

The basic transition system can be extended by introducing labels. Such labels are attached to transitions between the states of the abstract machine represented by the extended transition system. Such an extended transition system is called a labeled transition system (LTS).

Definition 2 (Labeled Transition System)
An LTS is a tuple (S, T, A), where S is a set of states, $T \subseteq (S \times A \times S)$ is a labeled transition relation and A is a label set.

Labels are used to reflect information related to transitions between states. For example, labels might model the input of values, which trigger a state transition. Labels can also represent conditions which need to be fulfilled in order to perform a transitions or reflect actions of the system when performing a transition from one state to another one. In this scenario, a distinguished event τ is often used to model internal actions of a system. If $s, s' \in S$ and $\alpha \in A$ and $(s, \alpha, s') \in T$ then we also write $s \xrightarrow{\alpha} s'$,

We further extend the definition of a labeled transition system to a timed labeled transition system by separating the set of labels into a label set and a time domain. The intuition is that from a state s either a labeled transition or a timed transition is possible.

Definition 3 (Timed Labeled Transition System)
A timed LTS (TLTS) is given by a tuple (S, T, A, D), where S is a set of states, $T \subseteq S \times (A \cup D) \times S$ is the timed transition relation, A is a label set and D is a time domain (such as $\mathbb{N}$ or $\mathbb{R}_{\geq 0}$) with A and D disjoint. We write $s \overset{d}{\rightsquigarrow} s'$ for $(s, d, s') \in T$ with $d \in D$. For each TLTS, we require the following two properties to be fulfilled:

- $\forall\ t_1\ t_2.\ s \overset{t_1+t_2}{\rightsquigarrow} t \longrightarrow \exists\, s'.\ s \overset{t_1}{\rightsquigarrow} s' \wedge s' \overset{t_2}{\rightsquigarrow} t$
- $\forall\ t.\ (s \overset{t}{\rightsquigarrow} s' \wedge s \overset{t}{\rightsquigarrow} s'') \longrightarrow s' = s''$

The first requirement ensures that time steps can be arbitrarily split into two consecutive timed transitions, i.e., there is a state s' available for any t_1 and t_2. The second requirement demands that the passage of time is deterministic.

2.3.2 Bisimulations

The formal notion of bisimulation [Mil89] is a binary relation defined on the states of an LTS. It relates two states if equivalent behavior is possible from the two states. Depending on the actual definition, bisimulation relations can also introduce abstraction. Furthermore, bisimulations can also be defined on timed labeled transition systems.

The least abstract version of bisimulation identifies two states s and t as being *bisimilar* if any outgoing labeled transition originating in s can be simulated by a transition with the same label originating from t and leading to a state which is again in the bisimulation relation and vice versa. Such a relation is called *strong bisimulation*.

Definition 4 (Strong Bisimulation)
A relation $R \subseteq S \times S$ is called a strong bisimulation on an LTS (S, T, A) if the following properties hold: For all $(s, t) \in R$ and $\alpha \in A$:

1. *If $s \xrightarrow{\alpha} s'$, then there is a t' with $t \xrightarrow{\alpha} t'$ and $(s', t') \in R$.*
2. *If $t \xrightarrow{\alpha} t'$, then there is a s' with $s \xrightarrow{\alpha} s'$ and $(s', t') \in R$.*

To allow for a certain degree of abstraction when relating processes, the notion of *weak bisimulation* allows us to abstract from 'internal behavior'. Internal behavior is modeled by a distinguished event, which is often realized as τ. The notion of *weak bisimulation* then abstracts from these state transitions by allowing an arbitrary number of these transitions before and after the visible transition that 'answers' a given labeled transition.

To define the notion of *weak bisimulation* we first define an extended step relation on an LTS (S, T, A), which abstracts from possible internal steps (denoted as τ) as $\longrightarrow_w \subseteq S \times A \times S$.Here, $\xrightarrow{\tau}^*$ is the reflexive and transitive closure of τ steps.

Definition 5 (Visible steps modulo τ)
1. *If $s \xrightarrow{\tau}^* s'$, then $s \xrightarrow{\tau}_w s'$.*
2. *If $s \xrightarrow{\tau}^* s_1$, $s_1 \xrightarrow{\alpha} s_2$ and $s_2 \xrightarrow{\tau}^* s'$ $(\alpha \neq \tau)$, then $s \xrightarrow{\alpha}_w s'$.*

Using this definition, we can now define the notion of *weak bisimulation*, which allows zero or more steps before the labeled transition that is used to simulate some other labeled transition. From the definition of the extended step relation (that abstracts from internal steps), it follows that if α in the definition below is an internal step, this transition can be simulated by zero or more internal steps.

Definition 6 (Weak Bisimulation)
A relation $R \subseteq S \times S$ is called a weak bisimulation on an LTS (S, T, A) if the following property holds: For all $(s, t) \in R$ and $\alpha \in A$

1. *If $s \xrightarrow{\alpha} s'$, then there is a t' with $t \xrightarrow{\alpha}_w t'$ and $(s', t') \in R$.*
2. *If $t \xrightarrow{\alpha} t'$, then there is a s' with $s \xrightarrow{\alpha}_w s'$ and $(s', t') \in R$.*

The notion of *weak bisimulation* can be extended to the setting of timed labeled transition systems as introduced above. The idea is to abstract from internal steps as in the definition of *weak bisimulation*. Furthermore, it is abstracted from individual time steps in the sense that a timed transition is allowed to be simulated using a number of timed transitions, which together represent the

same amount of time passing by. Moreover, between these timed transitions, an arbitrary number of internal transitions is allowed.

Using the definition of $\longrightarrow_w$, we define an extended transition relation $\longrightarrow_{wt} \subseteq S \times (A \cup D) \times S$ on a timed labeled transition system LTS (S, T, A, D). The extended transition relation allows to aggregate an arbitrary number of timed transitions into a single timed transition, which has a duration that equals the sum of the durations of the individual transitions and furthermore allows us to abstract from internal transitions.

Definition 7 (Aggregation of Timed Steps modulo τ)

1. *If* $s \xrightarrow{\alpha}_w s'$ *and* $\alpha \in A$, *then* $s \xrightarrow{\alpha}_{wt} s'$.
2. *If for all* $i \in \{0, \ldots, n\}$ *(for some arbitrary* $n \in \mathbb{N}$*):* $s_i \xrightarrow{t_i}_w s_{i+1}$ *with* $t_i \in D$ *and* $\sum_{i=0}^{n} t_i = t$, *then* $s_0 \xrightarrow{t}_{wt} s_{n+1}$.

Using the definition of this extended step relation we define the notion of *weak timed bisimulation* [dFELN99] as follows:

Definition 8 (Weak Timed Bisimulation)
A relation $R \subseteq S \times S$ *is called a weak timed bisimulation on a timed LTS* (S, T, A, D) *if the following property holds: For all* $(s, t) \in R$ *and* $\beta \in A \cup D$*:*

1. *If* $s \xrightarrow{\beta} s'$, *then there is a* t' *with* $t \xrightarrow{\beta}_{wt} t'$ *and* $(s', t') \in R$.
2. *If* $t \xrightarrow{\beta} t'$, *then there is a* s' *with* $s \xrightarrow{\beta}_{wt} s'$ *and* $(s', t') \in R$.

Using the different notions of bisimulation defined in this section, we can identify states s and t in a labeled or timed labeled transition system from which similar behavior is possible. Here, the notion of similarity depends on the level of abstraction that is inherent to the respective notion of bisimulation. The states s and t are called *strongly bisimilar* if there exists a relation R that is a strong bisimulation and if $(s, t) \in R$. If similarity is meant to abstract from possible internal actions and R is either a weak or weak timed bisimulation, then for $(s, t) \in R$ the states s and t are said to be weak or weakly timed bisimilar. When considering bisimulations in the context of process-algebras, for example, the states of an LTS are also called processes. If a transition labeled with α from such a process P to another process P' is possible, then P' is called an α-derivative of P'.

In this section, we have introduced labeled transition systems and the notion of bisimulation. We use these concepts as the foundation of a mechanized conformance relation between low-level languages and process-algebraic specifications. As a process-algebra, we use the formalism of Communicating Sequential Processes and its extension with time, Timed CSP which we introduce in the next section.

2.4 Communicating Sequential Processes (CSP)

In this section, we introduce the languages of Communicating Sequential Processes (CSP) and Timed Communicating Sequential Processes (Timed CSP). CSP is a language that belongs to the family of process algebras or process calculi. These formalisms target the high-level description and analysis of the interaction, communication and synchronization between a set of processes. For reasoning about specifications, algebraic laws are provided. Timed CSP can be regarded as a conservative extension of CSP by primitives to specify the timing behavior of processes. We start by explaining the syntax of CSP and subsequently discuss its operational semantics. Afterwards, we introduce the syntactical constructs to handle timing aspects of distributed processes and discuss how these are handled within the operational semantics of Timed CSP.

2.4.1 CSP

The language of Communicating Sequential Processes (CSP) [Hoa85] was first introduced by Tony Hoare in 1978 in a seminal paper [Hoa78]. The key idea underlying the formalism is to regard distributed systems as consisting of entities or processes, which behave or execute independently from each other but might choose to synchronize their behavior at some point in time and then proceed independently again until the next point of synchronization is reached. This observation already yields the main ingredients of CSP: *processes*, whose behavior is described in terms of instantaneous *events* that the respective process is ready to perform at some time of execution. Processes and the events they perform independently or (in case of a synchronization) together with other processes can then be observed by the environment of the processes. In the following, we first summarize the syntax of CSP. To give processes an unambiguous meaning, we discuss in detail the operational semantics that characterize the possible behaviors of processes in terms of firing rules that reflect the next step or steps a process is able to perform. In contrast to this operational small-step way of describing the possibilities of a process, denotational semantics for CSP give processes a meaning by mapping the syntactical description to a mathematical domain and thereby directly describing all the possible future behaviors of a process. We briefly present the main concepts of denotational semantics for CSP after our presentation of the operational semantics.

Syntax of CSP

The syntax of CSP reflects the main idea of the formalism: to focus on the interaction of sequential processes that may need to synchronize their behavior. A process can be identified by a name from the set of process variables $\mathcal{V}$. Processes are capable of performing events or synchronizing on events from an arbitrary alphabet Σ. However, there are two distinguished events $\tau \notin \Sigma$

and $\checkmark \notin \Sigma$ that processes are able to communicate but that do not belong to the alphabet. The first is used to describe internal actions of a process while the latter is used to model successful termination of a process. The set of events a process may perform or synchronize on at any time is specified using the operators shown in the following description of the CSP syntax. We focus on the operators relevant for this thesis. A detailed description can be found in [Ros10]

Definition 9 (Syntax of CSP)
Let a be an event from and A be a subset of the event alphabet Σ. The variable X ranges over the set of process variables $\mathcal{V}$.

$$P := STOP \mid SKIP \mid CHAOS \mid div \mid a \rightarrow P \mid x : A \rightarrow P_x \mid \\ P;\ P \mid P \,\Box\, P \mid P \sqcap P \mid P \mid_A \mid P \mid P \setminus A \mid X \mid \ldots$$

First, there are the predefined processes $STOP, SKIP$ and $CHAOS$ ($\notin \mathcal{V}$), which all model special types of process behavior. $STOP$ is not capable of doing anything else than deadlocking, $SKIP$ is not able to do anything but to terminate successfully. Successful termination is signaled to the environment by communication of the distinguished element $\checkmark$. The process $CHAOS(A)$ may perform any event from the set $A \subseteq \Sigma$ in any order or may refuse to do anything. Infinite internal behavior is denoted by the predefined process div. Using the *prefix* operator $\rightarrow$, it is possible to combine an event a with a process P, which yields a process that is able to first communicate the event a and afterwards behave like the process P. A generalization of *prefixing* is realized using the *menu choice* operator $x : A \rightarrow P_x$. A process constructed using this operator is capable of first communicating any event x from a specified set of events $A \subseteq \Sigma$ and afterwards behave like the process P_x. Using the operator for *sequential composition* $P;\ Q$, it is possible to combine two processes. The resulting process behaves like the first process and, if this process terminates successfully, then the resulting process behaves as the second process. The external choice operator $\Box$ offers the environment of a process the choice between two processes. The intuition is that if the environment chooses to synchronize on the initial event of the left process, then the combined process behaves as the left process and if the environment chooses to synchronize on the initial event of the right process, the combined process behaves like the right process. The *internal choice* operator $\sqcap$ has a similar behavior as the external choice operator but the decision about which process is chosen is made internally by the process. *Parallel composition* of processes (denoted by $|_A|$) allows two processes to run in parallel such that the processes can individually synchronize on any event outside of A but need to synchronize when performing events from A. When the hiding operator $P \setminus A$ is applied to a process P, all events from A that the process performs are translated to the internal τ events. Process variables X are used to model possibly mutually recursive processes. A mapping from the set of process variables to the set of possible CSP processes is used to associate process variables with CSP processes. In the next section,

we explain the informally introduced behavior of processes in more detail by discussing the operational semantics of CSP.

Operational Semantics

In this section, we give an overview of the operational semantics of CSP. The operational semantics describe the behavior of a CSP process in a small-step manner using inference rules (see Section 2.1). This motivates the interpretation of a CSP process as defining an LTS. The nodes of the LTS are given by process descriptions, while the transitions are labelled with events that are possible between two nodes (or processes). The rules of the operational semantics define which transitions are possible[1] from a given node. Our presentation of these rules follows [Sch99].

The process *STOP* is not able to do anything. Therefore no inference rule is given for *STOP*. Successful termination is modeled using the basic process *SKIP*:

$$\frac{}{SKIP \xrightarrow{\checkmark} STOP}$$

The process signals successful termination to the environment by communicating the distinguished event ✓, which only *SKIP* can perform. The operational behavior of the *prefix* operator is defined by the next inference rule.

$$\frac{}{(a \to P) \xrightarrow{a} P}$$

The event a is offered to the environment and after the environment decides to synchronize on a the prefix process behaves like the process P. Given a set of events $A \subseteq \Sigma$, the *menu choice* operator generalizes the prefix operator by offering any of the events from A to the environment:

$$\frac{}{(x : A \to P_x) \xrightarrow{a} P_a} \; a \in A$$

Once the environment has decided to synchronize on an event a, the constructed process behaves (dependent on the respective event a) like the process P_a.

The *external choice* formalizes that the environment of a process can decide between two possible processes:

$$\frac{P \xrightarrow{\mu} P'}{P \,\Box\, Q \xrightarrow{\mu} P'} \; \mu \neq \tau \qquad \frac{P \xrightarrow{\tau} P'}{P \,\Box\, Q \xrightarrow{\tau} P' \,\Box\, Q}$$

[1] these rules are also called firing rules

$$\frac{Q \xrightarrow{\mu} Q'}{P \,\square\, Q \xrightarrow{\mu} Q'}\ \mu \neq \tau \qquad \frac{Q \xrightarrow{\tau} Q'}{P \,\square\, Q \xrightarrow{\tau} P \,\square\, Q'}$$

The rules on the left formalize that internal events do not resolve the *external choice* operator. If one of the processes can evolve in terms of internal actions, the environment can still choose between possible visible events afterwards. The *external choice* is resolved in favor of one of its subprocesses if the respective process is able to communicate a visible event.

In contrast to external choice, *internal choice* is resolved without the cooperation of some environment:

$$\frac{}{P \sqcap Q \xrightarrow{\tau} P} \qquad \frac{}{P \sqcap Q \xrightarrow{\tau} Q}$$

The combined process is either resolved in favor of the left or the right process by firing a τ transition.

Sequential composition of processes is formalized by two firing rules, which reflect that the first process involved in the composition can evolve and if the process terminates successfully, the *sequential composition* behaves as the second process.

$$\frac{P \xrightarrow{\mu} P'}{P;\ Q \xrightarrow{\mu} P';\ Q}\ \mu \neq \checkmark \qquad \frac{P \xrightarrow{\checkmark} P'}{P;\ Q \xrightarrow{\tau} Q}$$

Note that the communication of the $\checkmark$ event (which signals successful termination to the environment) by the first process P is transformed to an internal transition.

The first two inference rules concerned with the *parallel composition* operator require the two subprocesses involved to synchronize on the communication of all visible events from the set $A \subseteq \Sigma$ and further to synchronize on termination.

$$\frac{P \xrightarrow{\mu} P' \quad Q \xrightarrow{\mu} Q'}{P \mid_A \mid Q \xrightarrow{\mu} P' \mid_A \mid Q'}\ \mu \in A^{\checkmark}$$

Events outside of the synchronization set A can be performed independently (this includes internal events).

$$\frac{P \xrightarrow{\mu} P'}{P \mid_A \mid Q \xrightarrow{\mu} P' \mid_A \mid Q}\ \mu \in \Sigma \setminus A^{\checkmark} \qquad \frac{Q \xrightarrow{\mu} Q'}{P \mid_A \mid Q \xrightarrow{\mu} P \mid_A \mid Q'}\ \mu \in \Sigma \setminus A^{\checkmark}$$

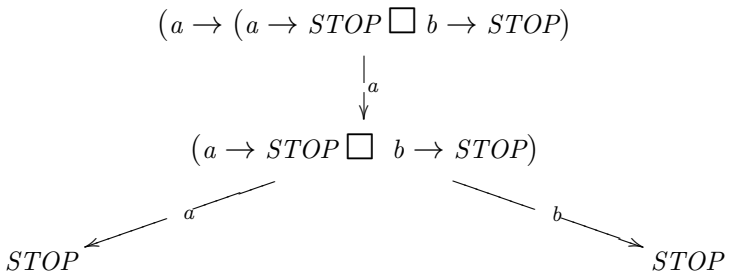

Figure 2.1: Labeled Transition System of a Process

This definition of parallel composition requires processes to terminate together, by synchronizing on the ✓ event. A different solution is given in [Ros05] where each process is allowed to terminate individually but the entire construction terminates when both processes have (this semantics is called Ω semantics).

The firing rules for the *hiding* operator formalize that all the events from the hiding set A are translated to the internal event τ, i.e., the process P can perform these events without synchronizing with the environment.

$$\frac{P \xrightarrow{\mu} P'}{P \setminus A \xrightarrow{\mu} P' \setminus A}\ \mu \notin A \qquad \frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A}\ a \in A$$

All events the process is able to perform which are not in A are under the control of the environment (except for the successful termination event ✓).

Variable assignment which is used to realize possibly mutually recursive processes is formalized using the following firing rule:

$$\frac{}{X \xrightarrow{\tau} asg(X)}\ X \in \mathcal{V} \qquad \text{with} \quad asg : \mathcal{V} \Rightarrow CSP$$

A process variable is replaced by its associated CSP process. This is reflected by an internal transition.

As explained in the beginning of our presentation of the operational semantics of CSP, using the presented inference rules, a CSP process can be interpreted in terms of a labeled transition system. Figure 2.1 gives a small example which shows the LTS as defined by the process $a \rightarrow (a \rightarrow STOP \,\Box\, b \rightarrow STOP)$.

Denotational Semantics

Several denotational semantics for CSP are well investigated and are the basis for refinement-based specifications. In this section, we briefly introduce the three most commonly used denotational models for CSP: the traces model,

the stable failures model and the failure divergence model. A detailed discussion of further semantic models for CSP and the connections between them is given in [Ros10] Starting with the traces model, each of the models adds more semantic information to the respective models.

The Traces Model The traces model is the basic denotational model for CSP. It denotes the semantics of a given process using the possible traces of externally visible events a process might engage in. A trace is basically a list of visible events that a process is able to communicate.

The Stable Failures Model An obvious short-come of the traces model is that it only supports the analysis of safety properties, i.e., that the process cannot engage in any events it is not supposed to engage in. The stable failures model overcomes this limitation. It builds on the traces model and records for every stable point within the evolution of a process the events that the process can refuse to synchronize on. A stable point is a point within the execution of a process from which no τ-step is possible.

The Failure Divergence Model The failure divergence model supplements the stable failures model with information about the possibility of a process to diverge, i.e., to perform an infinite number of τ-steps from a certain point of process evolution.

Tools for CSP

Over the years, a variety of industrial strength tools have been developed for CSP. These tools can be generally categorized as belonging to one of the following three categories:

- **Animators** can be used to interactively examine specifications given in CSP. The user can analyze a given specification by stepping through it. The tools ensures that the rules of the operational semantics are respected when stepping through a specification. When there is more than one event possible at a certain point of simulation, i.e., the animated process contains external or internal choices, the user might decide how the choice is resolved. Animation facilities for CSP are available in ProBE [Ros05] and ProB[LF08].
- **Model checkers**, in general, verify that a given model satisfies a formula given in a temporal logic. A model checking approach for CSP is realized in ProB [LF08]. Temporal logic formulae can be used to specify whether a given event is enabled or not at a certain point within the progress of a process.
- **Refinement checkers** for CSP are based on the denotational semantics and its notion of refinement. For CSP a well known refinement checker

is the Failure Divergence Refinement Checker (FDR2 [GRA05]). FDR can establish refinement relations automatically without any interaction of the user. Its main limitation is that it can only check finite state models. A refinement checker that is capable of analyzing infinite state processes is the CSP Prover [IR05]. However, since it is based on the interactive theorem prover Isabelle/HOL, refinement can in general not be analyzed automatically. Proving a refinement relation may need a lot of interaction from the user.

2.4.2 Timed CSP

To facilitate reasoning about the timing behavior of communicating sequential processes, CSP as introduced in the last section can be extended to a timed formalism. The extension we consider here is called Timed CSP [Sch99]. Timed CSP has a syntax similar to CSP and provides further operators which can be used to model timing behavior. We introduce the syntax of Timed CSP.

Syntax

Let a be an event from and A be a subset of the event alphabet Σ. The variable X ranges over the set of process variables $\mathcal{V}$. Let d be a value from the positive reals.

$$P := STOP \mid SKIP \mid a \rightarrow P \mid a : A \rightarrow P_a \mid P \,;\, P \mid P \,\Box\, P$$
$$\mid P \sqcap P \mid P \mid_A \mid P \mid P \setminus A \mid P \,\triangle\, P \mid P \overset{d}{\triangleright} P \mid P \triangle_d P \mid X$$

The informal meaning of the basic processes and process operators shared with pure CSP remains unchanged. For these operators, the timing behavior is defined as explained below in the operational semantics. The intuition of the newly introduced *timeout* construct ($P \overset{d}{\triangleright} Q$) is that the timeout is resolved in favor of process P if the environment decides to synchronize on one of the initially possible visible events of P within d time units. The idea behind the *timed interrupt* operator $P \triangle_d Q$ is that the process P is able to terminate within d time units. If P does not signal successful termination using the distinguished event $\checkmark$, P is aborted and the constructed process behaves as process Q. Further processes can be defined based on these operators. For example, the $WAIT$ process that lets time pass by is defined as $WAIT\ d \equiv STOP \overset{d}{\triangleright} SKIP$. In the following section, we explain the meaning of the timing constructs by presenting the firing rules of the operational semantics for Timed CSP.

Operational Semantics

Similar to the semantics of CSP, the formal semantics of Timed CSP processes can also be defined in terms of an operational semantics using inference

rules. For Timed CSP these rules can be understood as defining a timed LTS $(TCSP, (\longrightarrow \cup \rightsquigarrow), \Sigma \cup \{\tau, \checkmark\}, \mathbb{R}_{>0}))$. In such a timed LTS two kinds of transitions are possible: event transitions (denoted $\longrightarrow$) and timed transitions (denoted $\rightsquigarrow$). Event transitions are instantaneous in the sense that no time passes by when such a transition fires while timed transitions model the passage of time explicitly.

The basic processes *STOP*, *SKIP* can let time arbitrarily advance:

$$\frac{}{STOP \overset{t}{\rightsquigarrow} STOP}\ t > 0 \qquad \frac{}{SKIP \overset{t}{\rightsquigarrow} SKIP}\ t > 0$$

Similarly, the processes built using the operators *prefix* and *menu choice* have the possibility of letting time advance for an arbitrarily long duration until the environment is ready to engage in the respective events.

$$\frac{}{a \to P \overset{t}{\rightsquigarrow} a \to P}\ t > 0 \qquad \frac{}{x : A \to P_x \overset{t}{\rightsquigarrow} x : A \to P_x}\ t > 0$$

The timing behavior of the sequential composition operator is realized so that in the first process, time can only pass by if the process is not currently able to terminate successfully:

$$\frac{P \overset{t}{\rightsquigarrow} P' \quad \neg\,(P \overset{\checkmark}{\longrightarrow})}{P;\ Q \overset{t}{\rightsquigarrow} P';\ Q}$$

This means that if the first process is ready to communicate the event ✓, it has to do so as soon as possible. Otherwise, the timing behavior of the process P is given by the respective rule that applies to its current process state.

A process built using the *external choice* operator can let time advance if both of its subprocesses can let time advance.

$$\frac{P \overset{t}{\rightsquigarrow} P' \quad Q \overset{t}{\rightsquigarrow} Q'}{P \,\Box\, Q \overset{t}{\rightsquigarrow} P' \,\Box\, Q'}$$

There is no timed rule for *Internal Choice*. Corresponding to the understanding that internal transitions happen as soon as they are available, *Internal Choice* is resolved immediately.

Parallely composed processes are allowed to let time pass by if each of them can:

$$\frac{P \overset{t}{\rightsquigarrow} P' \quad Q \overset{t}{\rightsquigarrow} Q'}{P \mid_A \mid Q \overset{t}{\rightsquigarrow} P' \mid_A \mid Q'}$$

A process constructed using the *hiding* operator lets time pass by, if the process P that the *hiding* operator is applied to is not able to currently perform an event a from the hiding set A:

$$\frac{P \overset{t}{\rightsquigarrow} P' \quad \forall\, a \in A.\ \neg\,(P \xrightarrow{a})}{P \setminus A \overset{t}{\rightsquigarrow} P' \setminus A}$$

Events from A which are turned into internal events need to be engaged in immediately.

In a *timeout* process, it is possible for the first process P to synchronize on one of its visible initial events within d time units and thereby resolve the timeout. Internal events of P do not resolve the timeout. If the specified timeout interval d is elapsed the timeout is resolved in favor of the second process Q by an internal transition.

$$\frac{P \xrightarrow{\mu} P'}{P \overset{d}{\rhd} Q \xrightarrow{\mu} P'}\ \mu \neq \tau \qquad \frac{P \xrightarrow{\tau} P'}{P \overset{d}{\rhd} Q \xrightarrow{\tau} P' \overset{d}{\rhd} Q} \qquad \frac{}{P \overset{0}{\rhd} Q \xrightarrow{\tau} Q}$$

The specified timeout interval d is decreased whenever the first process lets time pass by.

$$\frac{P \overset{t}{\rightsquigarrow} P'}{P \overset{d}{\rhd} Q \overset{t}{\rightsquigarrow} P' \overset{d-t}{\rhd} Q}\ t \leq d$$

The behavior of the *timed interrupt* operator is that if the process P is able to signal successful termination by communication of the event $\checkmark$ within the specified interval, then the operator is resolved in favor of P. The process P can engage in any events within the specified timeout interval d. However, if the process P is not able to communicate $\checkmark$ within d time units, then the operator is resolved in favour of the second process Q. This behavior is defined by the following rules:

$$\frac{P \xrightarrow{\checkmark} P'}{P \triangle_d Q \xrightarrow{\checkmark} P'} \qquad \frac{P \xrightarrow{\mu} P'}{P \triangle_d Q \xrightarrow{\mu} P' \triangle_d Q}\ \mu \neq \checkmark \qquad \frac{}{P \triangle_0 Q \xrightarrow{\tau} Q}$$

As in the rule for *timeout*, in the rule for the *timed interrupt* the specified interrupt interval d is decreased whenever the first process is able to let time pass by.

$$\frac{P \overset{t}{\rightsquigarrow} P'}{P \triangle_d Q \overset{t}{\rightsquigarrow} P' \triangle_{d-t} Q}\ t \leq d$$

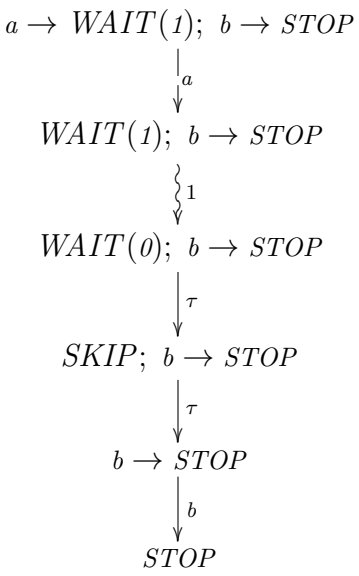

Figure 2.2: Timed Labeled Transition System of a Process

As demonstrated in the previous section for CSP processes, the operational semantics of Timed CSP can be used in a similar fashion to interpret a Timed CSP process in terms of a timed LTS. Figure 2.2 gives an example of such an interpretation. It visualizes the simple process $a \to WAIT(1);\ b \to STOP$. The process can communicate an a and afterwards waits for one time unit before communicating the event b. Then the process stops. Not depicted are the possible time loop transitions, i.e., the transitions possible when the process is able to communicate a and b and after it has stopped. Note that the second transition denotes a timed transition. Such a timed transition symbolizes arbitrarily many intermediate processes because a value from the reals is used here.

Tools for Timed CSP

Tool support for the real-time extension of CSP is also available. The FDR2 model checker can be used for analyzing Timed CSP specifications with discrete timing values [ALOR12]. The approach to automatic verification is based on digitization of real-valued Timed CSP processes. Specifications use the special event *tock* to model the passage of time. Specifications can then be analyzed using the notion of refinement.

The tool HORAE [DHSZ06] uses translations from Timed CSP processes to Constraint Logic Programming problems to enable reasoning about Timed CSP processes. The tool claims to support automatic proofs of safety properties, deadlock-freeness and also timewise refinement relationships.

Animation of Timed CSP processes is also available [FAGNR12] as an extension of the ProB tool. The animator exploits the observation that Timed CSP is closed under rational time. Rational time valued Timed CSP processes

are then translated to a representation in Prolog. The firing rules of the operational semantics of Timed CSP are implemented using Prolog and used to determine the available actions of a process.

In [Göt12] a verification environment for Timed CSP is presented, which is based on a mechanization of the operational semantics of Timed CSP in the theorem prover Isabelle/HOL. Processes can be analyzed either using a timed variant of Hennesy-Millner logic or by showing equivalence to other (possibly simpler) processes using the notion of bisimulations. While the latter is semi-automatic, also transformations for small instances of Timed CSP processes to the model checkers FDR2 and UPAAL are available, which facilitate automatic analysis.

The verification environment presented in the remainder of this thesis can be used for conformance proofs between programs given in low-level languages and specifications given in a process-algebra as presented in this section. To cope with the complexity of such proofs on the one side, and to ensure the overall correctness of our verification environment on the other side, we chose to mechanize our environment using the interactive theorem prover Isabelle/HOL.

2.5 Isabelle/HOL

The theorem prover Isabelle [NPW02] is a general framework for machine-assisted proofs based on the LCF approach [Gor00]. The system is built around a trusted kernel of inference rules which is responsible for all manipulations of a given logical context. Based on the inference rules implemented in this kernel, further rules can be composed and implemented using ML [Pau96]. By breaking down complex rules to applications of the basic inference rules within the logical kernel, soundness of logical contexts is preserved.

The system is generic in the sense that it can be instantiated with different logics. Isabelle's meta-logic is an intuitionistic higher-order logic and can be used to instantiate the system to a variety of concrete object logics, for example first order logic (FOL), Zermelo-Fraenkel set theory (ZF) or Higher Order Logic (HOL), which we use in this thesis.

2.5.1 Proofs in Isabelle

Logical reasoning in Isabelle is achieved by applying inference rules on the level of the meta-logic. The basic operators on the level of the meta-logic are implication ($\Longrightarrow$), equality ($\equiv$) and universal quantification ($\bigwedge$). These are not to be confused with the logical operators of an object logic, for example when using the instantiation to HOL, the symbol $\longrightarrow$ denotes the classic implication of HOL, while $\Longrightarrow$ is used as implication on the meta-logic in inference rules, for example. Therefore $\Longrightarrow$ cannot appear in HOL formulas. In general, such

inference rules have the following structure:

$$[\![P_1\ ;\ \ldots\ ;\ P_l]\!] \Longrightarrow Q$$

Such a rule is understood as an inference rule with the premises P_1 to P_l and the conclusion Q. The premises and the conclusion are given as terms in the object logic. The brackets are a short hand for nested entailment, i.e., the inference rule is equivalent to

$$P_1 \Longrightarrow \ ... \ (P_{l-1} \Longrightarrow (P_l \Longrightarrow Q))$$

Inference rules might contain schematic variables. These are variables denoted with a question mark, i.e., $?y$. Once a theorem is proved successfully all free variables are turned into schematic variables and can be instantiated or unified when using the theorem as an inference rule. One or more inference rules can be combined to *tactics* which are used to manipulate logical contexts, also called proof states. These have the following structure:

$$\bigwedge x_1\ \ldots\ x_m.\ [\![A_1\ ;\ \ldots\ ;\ A_n]\!] \Longrightarrow C$$

Here, the variables x_1 to x_m are fixed meta-variables. The goal represented by this proof state is to establish that given the assumptions A_1 to A_n, the conclusion C can be established. When applying inference rules to proof goals, the concept of *unification* is used to instantiate schematic variables with variables from the proof state. In cases where this cannot be done automatically, the instantiation can also be done manually. Reasoning in Isabelle is mainly done in a backward manner, i.e., the conclusion of a theorem is manipulated until it can be established from the assumptions. This means that when applying the inference rule from above to the proof state mentioned above using the application of the rule `rule` in `apply(rule` *inferenceRule*`)`, the conclusion Q of the inference rule is unified with the conclusion C of the proof state. The resulting proof obligations are again of the form of proof states and require to establish that each of the premises P_1 to P_l of the inference rule can be established using the assumptions A_1 to A_n from the proof state shown above. However, also forward reasoning is supported. When applying the proof rules `drule` and `frule`, the unification system attempts to unify the premises of the inference rules with some assumptions given in the proof goal and either replaces these assumptions with the conclusion of the inference rule (in case of `drule`) or keeps the assumptions and adds the conclusion of the inference rule to the assumptions. A combination of forward and backward reasoning is realized using the proof rule `erule`. Application of rules can be combined into tactics which can be implemented using the ML interface of Isabelle. Furthermore, *tactics* can be combined into *tacticals* in order to provide even more automatization. Predefined tactics provided by the Isabelle system include the simplifier (`simp`) which uses term rewriting and simplifies the proof state by the subsequent application of rules from the so-called simp set. The users can declare such rules, but caution has to be taken because application of rules might not terminate. The tactic `clarify` does not have this problem because

it applies only rules which might not lead to circular application of inference rules. In-between these tactics is the tactic `clarsimp` which applies modest rewriting.

Automatic theorem proving techniques are also available as proof methods. The `auto` method implements simplification together with classical reasoning and the method `blast` provides a tableaux reasoner. Furthermore, Isabelle provides an interface for external automatic theorem provers. Entire proof goals can be delegated to fully automated external provers like E [Sch02], SPASS [WDF+09] or Z3 [DMB08]. The goals are translated to the format of the respective prover and if the prover is able to find a proof, this proof can be reconstructed within the Isabelle framework. Therefore, proofs done by external provers do not introduce the risk of inserting logical inconsistencies into proofs.

In Section 5.4.1 on page 75, we explain a concrete proof using Isabelle/HOL in detail.

2.5.2 Types, Sets, and Functions in Isabelle/HOL

Currently, the most mature object logic for the Isabelle framework is Higher Order Logic (HOL). Besides the basic types and definition principles within the instantiation of Isabelle to HOL, a broad library of theories is provided. These theories formalize a variety of concepts that are well suited to aid the formalization of concepts from the area of computer science and can be used as the basis for further theory developments. In this section, we briefly present the main ingredients of HOL used within this thesis. For a thorough introduction we refer to the official tutorial [NPW02], which introduces Isabelle/HOL using the following equation: HOL = Functional Programming + Logic. The idea behind this is that in Isabelle/HOL, function definitions are reminiscent of functional programming in the sense that currying and pattern matching are used. Properties of functions operating on datatypes can then be specified and analyzed using the logical capabilities of higher order logic. Higher order logics allow quantification over predicates and functions of any order, i.e., predicates over predicates over predicates and so on.

Types

The basic types defined in Isabelle/HOL are truth values and natural numbers. Function types are defined using the infix notation $\Rightarrow$. For example, `nat` $\Rightarrow$ `int` defines a function type. Note that $\Rightarrow$ represents total functions. Type variables like $'a$ are defined as known from ML and can be used in conjunction with function types to define polymorphic types.

Further datatypes can be inductively defined using the command `datatype`. An important datatype in computer science is the type of **lists**. In Isabelle/HOL lists are defined as follows:

```
datatype 'a list = Nil                          ("[]")
                 | Cons "'a" "'a list"          (infixr "#")
```

Here, the definition of a list is parametric in the type variable $'a$. Therefore, list elements can be of any fixed type. For such datatype definitions certain lemmas are automatically generated and proved. This includes induction rules for performing inductions over the respective datatype as well as lemmas for case analysis. The Isabelle/HOL theory for lists offers a rich variety of functions and constructors for the definition and manipulation of lists.

HOL provides a constructor ($\times$) for the definition of pairs. Using the projection functions *fst* and *snd* a pair can be projected onto its left or right part. **Tuples** are defined based on pairs. For the projection of tuples onto certain elements Isabelle/HOL supports advanced pattern matching concepts. To further facilitate the usage of compound types, the type of **records** is provided. A record is defined as in programming languages consisting of possible n fields. Each record field has its own type. Even though records correspond to tuples and are internally defined based on pairs, for practical purposes they have certain advantages. Fields of records have dedicated names which are also used as selector names for projections onto the respective field. Furthermore, records can be extended. A record type is defined using the command `record` in the following manner:

```
record state = x :: int
               y :: nat
```

The record type is often used to formalize the concept of a state as known from the semantics of programming languages. In our example the record consists of two fields with the names x and y. Using the `definition` command a constant of type state can be defined:

```
definition s1 :: state where
  "s1 ≡ (|x = 1, y = 2 |)"
```

A record can also be *updated* point-wise using the notation `:=`, so the expression $s1(|\ x := 0\ |)$ leads to a record that consists of the fields x with value 0 and y with value 2.

Functions

Non-recursive functions can be defined in Isabelle/HOL using the command `definition`. For anonymous functions the λ-operator is available. Recursive functions can be defined as common in functional programming languages using pattern matching. For example, the command `primrec` supports the definition of primitive recursive functions. The following definition, for example, defines a function which reverses a list:

```
primrec rev :: "'a list => 'a list" where
  "rev [] = []" |
  "rev (x # xs) = (rev xs) @ (x # [])"
```

For logical consistency, functions in Isabelle need to terminate. For elementary function definitions, Isabelle is able to prove termination automatically by utilizing well-founded relations. Function updates can be defined using the directive `:=`. While in the example above we used a record to formalize the notion of a state, to explain function updates we now use a function from a type of variables to the natural numbers to formalize the notion of a state:

```
primrec state :: "variables ⇒ nat" where
  "state x = Suc 0" |
  "state y = Suc 0" |
  "state z = Suc 0"
```

A pointwise function update for the argument x is then written as $(state(x := 2, z := 2))$. The resulting (or *updated*) function then agrees with the original function on all parameters (y in this example) except the ones mentioned in the update (x and z in the example).

Functions which cannot be defined using the command `primrec` can be defined using either `fun` or `function`. When `fun` is used, the theorem prover attempts to prove termination automatically by utilizing lexicographical orderings. For `function`, a well-founded relation needs to be provided and it further needs to be established that recursive function invocations decrease the arguments of each recursive invocation.

Inductive and Coinductive Sets

Isabelle/HOL provides sophisticated support for the specification of inductively defined structures. Inductive definitions consist of introduction rules as illustrated in the following standard example (from [TN13]):

```
inductive_set even:: "int set" where
    "0 ∈ even"
  | "n ∈ even ⟹ (Suc (Suc n)) ∈ even"
```

Here, the set of even number is defined using the directive `inductive_set`. Once an inductive definition is given, the Isabelle system automatically proves a variety of theorems about the structure at hand. These can be used for induction proofs or when, for example, a case analysis is required. As a dual concept to inductive definitions where introduction rules as given above are interpreted in terms of their smallest fixpoint, coinduction [JR97] [Gle05] takes the dual approach by interpreting such rules as defining a greatest fixpoint. If the directive `inductive_set` in the definition given above is replaced by the directive `coinductive_set` the Isabelle system realizes exactly this approach. In a similar fashion as for the inductive definition, certain theorems are automatically established. Of particular importance is the resulting coinduction scheme which can be used to establish that some element is contained in an coinductively defined set.

Local Proof Contexts

When conducting non-trivial formalization projects, it is often the case that the need for parametrization arises. Locales in Isabelle/HOL [Bal06] are a flexible mechanism to realize parameterized proof contexts in such formalizations. The intuition is that function constants and assumptions about them can be defined and fixed in an abstract manner and referred to using the locales name. Proofs can then be done within such a named context and refer to the fixed types and assumptions. It is also possible to introduce a structure on locales by extending existing locale contexts with additional assumptions. Furthermore, such contexts can be instantiated with concrete values so that proofs on the concrete level can inherit facts established on the abstract context.

As an example, we take the definition of a group known from algebra. We define a locale named *group* in the following way:

```
locale group =
  fixes product  :: 'a ⇒ 'a ⇒ 'a (infixl "×" 50)
    and   inverse :: 'a ⇒ 'a
    and   neutral :: 'a
  assumes associative: (a × b) × c = a × (b × c)
    and left-inverse : (inverse a) × a = neutral
    and left-neutral : neutral × a = a
```

Here we specify the elementary assumptions about a group. By using the type variable *'a* the definition is parametric in the type of group elements. On the arbitrary but fixed type a mapping called *product* is assumed which maps two elements of the group to another group element and can be referred to by using the concrete symbol $\times$. The function *inverse* is of type $'a \Rightarrow' a$. Furthermore, the neutral element is fixed as being of type $'a$. Assumptions about the fixed constants can then be made in the `assumes` block of a locale's definition. Here, it is described that the product is associative and that the *inverse* element, if multiplied from the left, is mapped to the neutral element which in turn yields the identity function if multiplied with any group member.

Using the directive `in` within the declaration of a theorem, the respective theorem is established within the context of the locale. Within the proof it can then be referred to the fixed constants and assumptions. Finished theorems are stored within the locale context and can then be built on when proving further theorems in the context. It is also possible to define a whole context block using the following construction:

```
context group
begin
...
end
```

A strength of the `locale` concept is that it becomes possible to fix assumptions which are used in one or more theorems as a group and use them in theorems without always having to mention them explicitly in a theorems assumptions. Another strength is that a hierarchy can be established on locales.

```
locale abeliangroup = group +
  assumes commutative : a × b = b × a
```

In this definition, the locale context from above is extended to an abelian group by adding the requirement that the product is commutative. In such an extended group all the theorems established in the context that the extended locale builds on can be referred to.

To make use of the abstract results of a locale on a concrete level, the directive `interpretation` is used. For example, a minimal instantiation can be obtained as follows:

```
interpretation concrete: abeliangroup
apply(unfold locales)
apply(auto)
done
```

When instantiating such an abstract definition it is required to show that the assumptions of the locale actually hold for the specified constants and functions.

In this section, we have introduced the interactive theorem prover Isabelle/HOL, which we use to mechanize the theories discussed in this thesis. Going this way, we can cope with the complexity inherent to the conformance proofs in our setting. The interactive prover ensures that corner cases cannot be overlooked and also provides automatization of proof steps on the level of the underlying logic HOL.

2.6 Summary

In this chapter, we have introduced the basic concepts used within this thesis. We have introduced the proof rules for a Hoare-calculus targeting a basic sequential high-level language. In this thesis, we construct a Hoare calculus for a low-level language. A typical example of such a language is the intermediate representation of the LLVM framework which we have briefly presented in Section 2.2. In Section 2.3, we have revisited the definitions of transition systems and labeled transition systems and given the definitions of bisimulations and timed bisimulations. These can be used to establish that equivalent behavior is possible from states within a labeled transition system. Based on the presentation of the operational semantics of CSP, we have presented a real-time extension of classic CSP, called Timed CSP. It is our goal to formally define concepts that build on these already complex definitions and to furthermore create relations between them. To cope with the complexity and enable mechanized proofs, we use the theorem prover Isabelle/HOL which we have introduced in Section 2.5.

In the next chapter, we discuss related work and then in Chapter 4, we give an informal motivation and presentation of the envisioned framework presented in this thesis.

3 Related Work

In this thesis, we present our mechanized verification environment for low-level implementations of embedded real-time systems. Furthermore, our verification environment proposes a method for establishing conformance between different levels of abstraction as typically found in embedded development approaches. In particular, it is the goal of this verification environment to investigate the interface between the abstract level of process algebraic specifications and the concrete level of unstructured programming languages. To realize such a verification environment, we especially focus on the formal treatment of the low-level language. We provide a formal notion of conformance based on bisimulations to relate the implementation level with the level of the abstract specifications. There, we build on an existing mechanization of the process algebra Timed CSP [Göt12]. For the abstract level, we provide a modeling approach for reactive systems that adapt their behavior to stimuli from the environment. This makes our entire verification environment applicable to this class of systems.

The following discussion of related work is structured along these core aspects. We start with a discussion of formalizations and mechanizations of programming languages in theorem provers and focus on low-level and real-time languages in Section 3.1. Subsequently, in Section 3.2, we examine approaches that have the goal of formally relating different levels of abstraction. We mainly focus on approaches which aim at bridging the semantic gap, i.e., approaches that connect different semantic domains like programming languages and specification formalisms. We finish the discussion of related work with a review of approaches concentrating on the specification and verification of adaptive systems in Section 3.3 and a brief summary in Section 3.4.

3.1 Formalization of Program Languages and Proof Calculi in Theorem Provers

The history of the Hoare calculus for high-level languages has shown that the usage of theorem provers for mechanized proofs about the soundness and completeness of proof systems for high-level languages is beneficial. Subtle cornercases, which might be overlooked in pencil and paper proofs cannot be left

out in mechanized proofs. Therefore, we mainly concentrate on formalizations of programming languages in theorem provers in this section.

3.1.1 High-Level Programming Languages

In [Sch97], a Hoare logic for a while language was developed and formalized using the theorem prover LEGO. There, the necessity of explicitly considering auxiliary variables in Hoare calculi is highlighted. Auxiliary variables allow to fix the values of variables in preconditions and define postconditions that depend on the fixed values. Using the presented proof calculus and the explicitly modeled auxiliary variables a formal comparison of Hoare logics and VDM-style proof systems is given. Our extended low-level logic follows this approach in the treatment of auxiliary variables.

In [Nip02], various Hoare logics for a typical while language are formalized using the theorem prover Isabelle/HOL. The formalized total correctness logic can cope with unbounded non-determinism and mutually recursive procedures. In our formalization of the low-level language extended to non-determinisim and total correctness, we followed the proposed proof strategy for completeness of the Hoare logic which utilizes the most general triple approach [Gor75]. The proof logic presented in [Nip02] is further extended to cope with mutually recursive procedures, which requires a small-step semantics for soundness and completeness proofs.

A comprehensive and powerful mechanization of a Hoare logic is presented in [Sch05]. The approach uses a small core language called SIMPL and provides a proof logic for this language. Even though the language consists of a small kernel only, it is capable of handling advanced concepts like abrupt termination, for example. Building on this core logic, a subset of the C programming language called C0 is formalized.

The discussed formalizations of high-level languages have influenced our formalization for low-level languages mainly concerning the total correctness logic and auxiliary variables. However, timing behavior and communication are not considered in these approaches. Apart from [Nip02], small-step semantics is also not considered.

A high-level language for the specification and implementation of concurrent real-time systems is presented in [Hoo94]. There, the assertions of classical Hoare-logic are extended to a real-time scenario. The approach is formalized using the theorem prover PVS [ORS92] in [Hoo98b] . The approach supports synchronous and asynchronous communication. Components of concurrent programs are assumed to have the property of maximal progress, i.e., every sequential component of a concurrent composition is assumed to have its own processing resources. This view is especially manifested in the proof rule for concurrent composition where the termination time (if the concurrent program terminates) is assumed to be the maximum of the execution times of the subcomponents. Our experimental rule for reasoning about distributed communicating low-level code mentioned in Section 6.5 adapts the respective

rules from [Hoo98b] in order to obtain a compositional proof system for distributed systems. Compared to our environment, the approach focusses on the high-level implementation of real-time systems. Furthermore, it does not consider conformance relation between implementations and process-algebraic specifications.

3.1.2 Low-Level Programming Languages

The first attempts to cope with unstructured code were extensions to the original Hoare logic for the structured while language [Bru81]. The approach uses conditional Hoare triples to cope with restricted or unrestricted jumps. These triples allow to use assumptions about label invariants given for an entire program. Floyds original proof system for control-flow graphs [Flo67a] also considers unstructured control flow.

Research in proof systems for low-level language was mainly motivated by the rise of Java bytecode and the concept of proof carrying code [Nec97] (PCC[1]) In [SU05] a compositional semantics and proof calculus for low-level languages is presented. In this work, classical Hoare triples are used for the specification of program behavior. The underlying idea is that low-level code can be viewed as having an implicit structure in terms of finite unions of code. This enables the construction of a compositional big-step semantics and proof calculus. An invariant covering all of the available code, i.e., covering also parts of the program that are not subject to evaluation of the given big-step execution, is not needed in this approach. The authors show that proofs established using a Hoare calculus for a classical while language can be compiled to the low-level proof system. While the authors have given the basic proof rules for a total correctness calculus, no formal proofs of soundness and correctness were published. Our formalized low-level language and related proof systems further extends the approach of [SU05] by unbounded non-determinisim in the presence of total correctness. Furthermore, we mechanize the proof systems for partial and total correctness using Isabelle/HOL and thereby obtain a mechanized proof environment, which also provides partially automated proofs on the level of the underlying logic. A mechanization of the proof calculus from [SU05] for the partial correctness case using the theorem prover Coq is described in [ANY12]. There, the proof system is used to verify security properties of cryptographic algorithms given in an assembler-like representation. Besides the conceptual extension of providing a total correctness logic for a language that allows unbounded non-determinism, our approach to the mechanized verification of low-level code also considers timing and communication behavior as well.

Further low-level semantics and related proof calculi are presented in [Ben05, AM01, MG07, BH06, JBK13]. The work of Appel [AM01] uses continuations in assertions. The work of Benton [Ben05] uses a collection of global label in-

[1]The idea of PPC is that a compiled version of a program (for example in bytecode) is supplied together with a proof of correctness. The receiver then checks the proof in order to be sure that the code can be trusted.

variants, like deBruins approach for goto statements [Bru81] . Behringer and Hofman [BH06] provide a logic that implicitly relates assertions to the specifications of surrounding method bodies to 'restrict' the potential of unrestricted jumps. Even though proofs and assertions in our logic are potentially more verbose than in the approaches mentioned above, the compositional nature of the logic facilitates the transfer of verification results from the logic to the level of the small-step semantics.

3.2 Formal Relationships Between Different Abstraction Levels

The probably most general approach to formal relations between different levels of abstraction is Hoare und He's initiative of 'Unifying Theories of Programming' [HH98]. The goal is to realize a holistic approach to programming language theory. The authors propose a semantic framework that provides general relations that connect different semantic approaches, i.e., denotational, operational and axiomatic semantics. Compared to our approach, the goal of the initiative is to bring different semantics together on a meta-level, while we focus on an approach that supports the verification of implementations given as low-level code and abstract specifications given in a process-algebra.

The ProCos 2 project [HHMO$^+$96] investigated the integration of formal methods into the different levels of abstraction that an embedded real-time development process may be divided into. On the most abstract levels, informal requirements are translated to formal specifications in a subset of the Duration Calculus (DC) [CH04] called DC implementables. To enable a seamless transition from these specifications to the implementation layer, an intermediate specification language SL [Sch94] is introduced. This language is based on the mixed term approach which allows to mix specification constructs with programming language commands. On the implementation layer, the approach is based on a real-time variant of the programming language Occam (a programming language that is closely connected to CSP) called PL [Fv93]. PL extends Occam by commands that facilitate the specification of upper bounds for computations. Implementations given in this real-time dialect of Occam are in turn transformed to machine code, which runs on transputers[2] [Bar78], or to a hardware representation (circuit diagrams). Correctness of resulting low-level code is ensured by the correctness of the compiler. The different layers of abstraction (called the ProCos-Tower) therefore cover the entire transformation chain from informal requirements down to hardware implementations. Mostly related to the work of this thesis are the steps from SL specifications to PL code and from there to machine code. SL specifications are transformed in a stepwise manner using predefined transformation rules. The correctness of these rules is established using the underlying state-trace readiness semantics. Correctness of the transformation from high-level code to low-level code is ensured by compiler correctness. In our approach, we assume that a Timed

[2] transputers are parallel processors using CSP-like communication mechanisms

CSP specification is refined to a sufficiently detailed specification that can be implemented using a high-level language of the developers choice. The implementation is then transformed to the low-level representation and shown to be correct with respect to the specification using the notion of bisimulations. The semantics of the specification and the low-level implementation is given in terms of a labeled transition semantics. While the transition from the specification to the implementation is not as structured and automatized as in the ProCos approach, our environment offers the possibility of integrating ad-hoc optimizations of low-level and high-level code representations. Furthermore, our approach is independent from a fixed high-level language since we aim to support a common low-level representation as the basis for verification.

The real-time refinement calculus presented in [Hay02] extends the original refinement calculus by adapting the standard refinement rules to the real-time context. In this calculus stepwise refinement is realized by extending Dijkstra's weakest precondition calculus with a so-called specification command [Mor88]. The real-time refinement calculus further extends the original calculus with rules that cope with timing related concepts of the target implementation language like delays and deadlines. Using the extended calculus, real-time specifications can then be refined to platform independent programs given in this axiomatized high-level language. Further refinement steps to low-level code representations are not considered. The B method [Abr96] and also its newer version called Event-B provide a method that supports software development from abstract specifications to the level of the implementation with formally funded transformations. However, real-time and also transformations to low-level languages are not considered.

Other refinement methodologies like CSP-OZ-DC [HO02], TCOZ [MD98] and Real Time-Z [Sü99] support the formal specification and refinement of real-time systems. However, even though these formalisms provide state-based descriptions of the behavior of a system, the refinement steps do not support refinement to representations in programming languages.

3.3 Adaptive Systems

In [JST+09], CSP is used to model self-adaptive applications. In the described scenario, a network of communicating nodes is assumed to learn from the behavior of other nodes. Behavioral rules of nodes are described by CSP processes which are communicated between the nodes and used within nodes to adapt the individual behavior. In contrast, our work is focussed on modeling entire adaptive systems and verifying properties of the modeled systems. Furthermore, refinement of systems is not mentioned in [JST+09].

Dynamic reconfiguration of systems is supported by the work presented in [ADG98]. Systems are described using the architecture description language (ADL) Dynamic Wright which is inspired by CSP. Reconfiguration of interacting components is modeled separately from steady-state behavior in a central specification. The models are connected by control events. The formal

semantics of the ADL is defined by a translation to CSP. Thereby, dynamic architectural properties can be verified on the level of CSP. Our approach supports distributed management of reconfiguration and connects the system model focussing on adaption behavior and the one focussing on functional behavior via CSP refinement. As a consequence, our models can be developed and analyzed in a stepwise manner. Furthermore, our approach supports refinement down to an implementation, which faithfully implements adaption as well as functional properties specified on the abstract level. In [MK96], the ADL Darwin is presented. The work focusses on architectural aspects of adaptive system design and does not consider the interaction with general functional system behavior. The semantics of reconfigurations is defined using the π-Calculus. While this enables a precise and unlimited way of modeling dynamic aspects of reconfiguration, it limits the use of automatic proof tools.

The work presented in [ASSV07] presents a development approach for adaptive embedded systems starting with model-based designs, which are mapped to a formally defined intermediate representation. As in our approach, adaption behavior is strictly separated from functional behavior. Adaptive systems are modeled in terms of interacting services, which communicate via shared variables. Adaptions are triggered by quality descriptions of data values. High-level specifications of services can be given a formal meaning by mapping them to transition systems that are connected by input and output channels in order to compose the adaptive system to be modeled. Verification can be performed by model checking and theorem proving. This is realized by embedding the respective representations of the transition system semantics into the proof tools. The main goals of the presented work are to provide verification mechanisms for adaptive system developments and to exploit the inherent structure of specifications of adaptive systems for more efficient proof procedures. Our approach aims to support development processes for adaptive systems with the powerful notion of CSP refinement and the mature proof tools developed for simulation, refinement checking and LTL model checking of CSP specifications. Furthermore, by using CSP as a formalism, we can use our conformance relation for the transfer of properties to the level of a possible low-level implementation.

In [KSKA09], Kosek et al study the capabilities of JCSP [Wel98] with respect to the implementation of adaptive systems. This work presumes that CSP is suitable for the specification, modeling and verification of such systems, but gives no hints on how to use CSP in this context. Instead, they cite [WBP06] where `occam`-π (which is based on CSP concepts but is quite different from CSP) is used to implement complex adaptive systems. In our work, we focus on CSP as described by Roscoe in [Ros05] and its tools FDR [GRA05] and ProB [LF08, PL08]. Thereby our approach is supported by a variety of semi-automatic tools for the analysis of adaptive systems on the specification level. Furthermore, using our conformance relation we are able to support the implementation of adaptive systems using a low-level language that requires no advanced concepts like the ones used in JCSP, for example.

3.4 Summary

In this chapter, we have reviewed work related to the concepts used in our approach to conformance proofs between process-algebraic specifications and low-level implementations. Following this, we have discussed formal modelling approaches for the class of adaptive systems.

In our environment, the low-level semantics and the related proof calculus are carefully chosen to support concrete proofs using bisimulations as the notion of conformance. This close interaction is a main motivation for our extension of the compositional low-level semantics and proof calculus from [SU05], which, in contrast to other approaches, enables us to relate arbitrary points within a program using the proof calculus. Furthermore, the usage of Hoare triples enables the transfer and reuse of established proof techniques and mechanization approaches from high-level languages to low-level languages. Considering the notion of conformance in our setting, namely weak timed bisimulation, we observe that most related approaches are built around notions of denotational refinement. While this enables conformance proofs in a more abstract manner, our operationally inspired notion of conformance facilitates mechanization in a theorem prover considerably. Regarding the work related to our approach for the specification and verification of adaptive systems, we summarize that our approach enables the use of established and efficient proof tools for CSP. Moreover, using our conformance relation, we can support the transfer of properties from the abstract level to the level of the low-level implementations.

In the next chapter, we give a detailed semi-formal presentation of our verification environment. Afterwards, we present the integrated concepts in full detail in the subsequent chapters.

4 A Mechanized Verification Environment for Real-Time Process Algebras and Low-Level Programming Languages

In this chapter, we introduce our verification environment for low-level code and conformance proofs between process-algebraic specifications and their low-level implementations. Our verification environment tackles a verification problem that typically arises when developing safety-critical embedded real-time systems: to establish conformance between the different levels of abstraction a system is described on. In general, a formalism that is well suited for describing behaviors on a certain level of abstraction takes a different perspective onto the system under consideration compared to a formalism describing another level. This difference is manifested in the different semantic domains of the involved formalisms, i.e., in the meaning of a given system description. The implication is that proofs of conformance, which establish that behaviors on different levels of abstraction correspond to each other, are complex. Ideally, such proofs are realized using a thoroughly defined notion of conformance. For a framework that supports conformance proofs between different levels of abstraction, this results in mainly two challenges:

- From a theoretical point of view, an approach to unambiguously relate the different levels of abstraction and thereby the respective representations of a system needs to be provided. That is, to provide appropriate semantic descriptions for the different levels of abstraction and unambiguous notions that allow to relate them.
- From a practical point of view, to deal with the unraveling of abstractions. That is, to cope with behaviors that are not described in detail on the higher level of abstraction, but are specified in more detail on the implementation level.

The goal of our verification environment is to support embedded development approaches. Software for embedded devices often needs to cope with restricted resources. Furthermore, low-level representations of embedded soft-

ware are also often subject to manual optimizations. The implication for our approach is that we choose not to overcome the gap between high-level representations and low-level code using a predefined (and verified) compilation method. Instead, we focus on the treatment of low-level code and methods to analyze programs on this low level of abstraction and then discuss how conformance can be verified.

In the next section, we discuss how our verification environment deals with the challenges mentioned above. The aim is to give an informal but detailed overview of the building blocks that are discussed more thoroughly in the subsequent chapters of this thesis. We motivate the conceptional design decisions taken in our framework and explain how it can serve as a verification environment for concrete conformance proofs.

After the introduction of our mechanized environment for conformance proofs in Section 4.1, we give a brief introduction to the research project VATES[1] [GBGK10] in Section 4.2. The development process considered in this project is a typical example of a development approach that requires formally funded conformance proofs between different levels of abstraction. Moreover, we motivate that the conceptual framework discussed in this thesis can be regarded as providing a theoretical foundation for the VATES development approach and furthermore also supplies mechanized tool support for conformance proofs in this setting.

4.1 Mechanized Proofs of Conformance in the Context of Low-Level Code and Process Algebras

In this section, we explain our mechanized verification environment for low-level code and proofs of conformance in more detail. We start by explaining our choices of formalisms and languages and then explain our strategy for conformance proofs in more detail.

4.1.1 Basic Design Considerations

A main design criterion for our approach is given by the decision to choose the level of low-level representations as the basis for the verification of implementation correctness. In our approach, this level of abstraction is used to formally connect the implementation level with the level of the abstract specifications. For the level of the abstract specifications, we choose the process-algebra Timed CSP to enable the concise specification of communication and real-time behavior. On this abstract level, the formal concept of bisimulations can be used to relate semantically equivalent processes. More precisely, the formal notion of weak timed bisimulation is employed to relate processes that describe the system to be developed on different levels of detail by abstract-

[1] Verification and Transformation of Embedded Systems

ing from concrete descriptions of behaviors. The advantage of bismulations as a conformance relation is that they can be verified on the basis of the operational semantics. This especially facilitates the mechanization in theorem provers, since the mechanization of operational semantics in theorem provers is well-understood. To obtain a uniform verification environment and to realize a seamless transition from the level of the abstract specifications to the level of the low-level implementations, we base the verification of conformance relations between implementations and their related specifications on the notion of bisimulations as well. Going this way, we obtain a consistent top down approach, starting with a simple system description focussing on the main behaviors that can be observed of a system. Then, on a less abstract level of description more details are added to the design. To establish semantical equivalence of the detailed description to the more abstract level, the added details are then considered to be internal to the system and it is shown that when abstracting from these internal events, the observable behavior of the two system descriptions can be considered to be semantically equal. Note that this notion of conformance is in contrast to the notion of refinement. There, conformance between different levels of abstraction is not only based on the abstraction from internal behaviors, but also on the concept of resolution of non-determinism[2].

In the context of real-time systems, semantical equivalence between different levels of abstraction means that equivalent values can be received and communicated at the same points of time, but different ways of computing the respective values might be used. This resembles the formal concept of weak timed bisimulation. Since we consider a low-level representation (of the implementation in a high-level programming language) as the basis for verifying implementation correctness, the internal behaviors that need to be abstracted from can be considered to provide a main source of complexity for conformance proofs.

For example, the calculation of a result in response to some value received from the environment might be specified on the abstract level using some mathematical function. On the level of the low-level representation, this behavior is given by a complex series of transitions that operate on the internal state of the program and occur between the reception of the input parameter and the communication of the result. In general, for the verification of low-level code, this means that we need to be able to relate arbitrary states during execution. States (or points within the evolution of programs or processes) of different levels of abstraction are related in terms of the values that are received or communicated starting from them and in terms of the amount of time that might pass by from these states. Establishing conformance with respect to the motivated notion of weak timed bisimulation, i.e., showing that a low-level implementation behaves semantically equivalent to its respective specification on the more abstract level therefore consists of three main parts:

[2]In the classical notion of refinement a representation on the lower level of abstraction is assumed to exhibit less non-determinisim than a more abstract one, but the two do not have to be equivalent modulo internal behaviors.

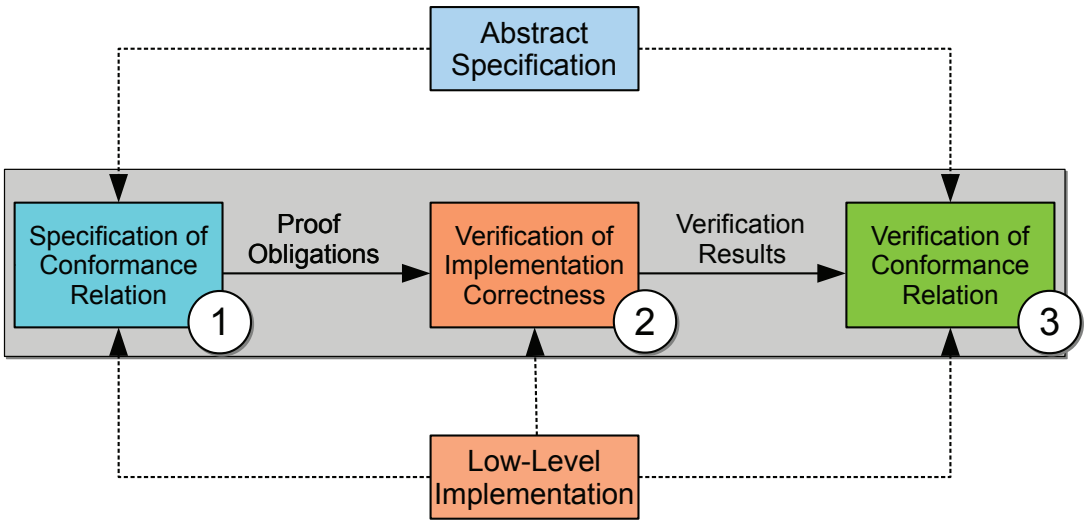

Figure 4.1: Verification Flow in our Verification Environment

1. The specification of a conformance relation identifying states from the abstract specification and states from the implementation is required. From the identified states only equivalent behaviors should be possible.

2. Proofs of correctness that low-level code computes specified results as required by the specified conformance relation, i.e., that the relation between values of variables at certain points of execution are as specified in the conformance relation. Furthermore, it is important that the amount of time which passes by in between adheres to specified values.

3. The specified conformance relation needs to be verified, i.e., the specified relation needs to fulfill the requirements of the chosen notion of conformance. This requires information about the possible behaviors of both the abstract specification and the low-level implementation. For the low-level implementation, we want to build on the correctness proofs mentioned in the previous step.

To apply this three-step-strategy (depicted in Figure 4.1), we first need to make the notion of weak timed bisimulation applicable in our setting. While the language and the semantics of Timed CSP already support the specification of functional and of timing behaviors, typical low-level languages, like the LLVM IR, in general provide no such capabilities. On the abstract level, we base conformance proofs on the operational semantics of Timed CSP. This semantics can be interpreted as a timed labeled transition system (TLTS). Furthermore, the notion of weak timed bisimulation is defined on timed labeled transition systems. We therefore provide a small-step semantics for low-level languages, which in turn can be interpreted as a timed labeled transition system.

Using the formal concept of timed labeled transition systems as an 'intermediate layer' to apply the notion of weak timed bisimulation yields a modular structure for our verification environment as depicted in Figure 4.2. We define bisimulations abstractly on timed labeled transition systems. To show that

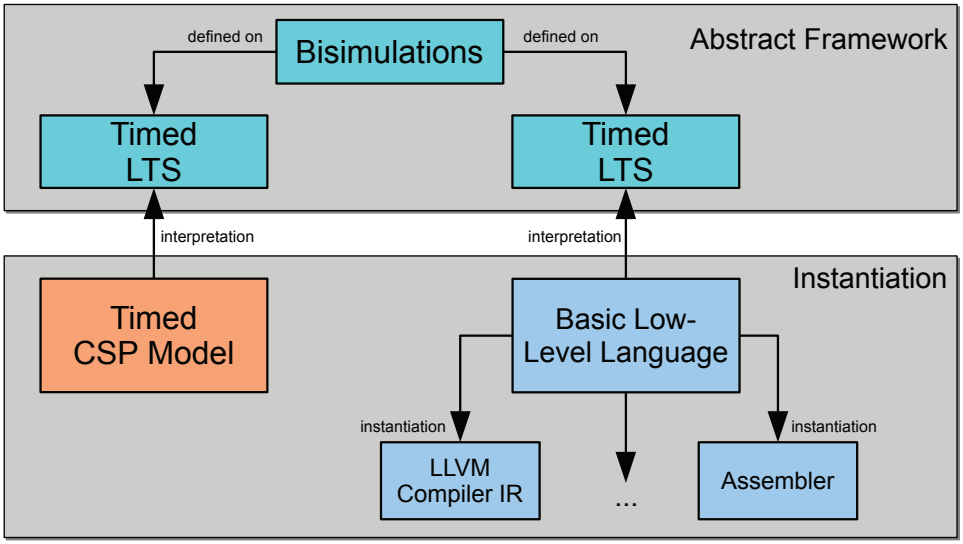

Figure 4.2: Modular Structure of the Verification Environment

a concrete program and process-algebraic specification are bisimilar, we then need to establish that its semantics can be interpreted as a TLTS. This way of modular specifications and instantiations is directly supported by the concept of locales in the Isabelle/HOL theorem prover. Furthermore, for low-level languages, we formalize a basic language that can be used as the basis for formalizations of more concrete and complicated unstructured languages. By realizing our verification environment in such a modular way, we can easily extend the environment to other languages and formalisms that can be interpreted as a labeled transition system. For example, we could use our low-level semantics and proof calculus as the basis for a formalization of an assembler language or a bytecode representation. Our mechanization in Isabelle/HOL could then be used as a basis for such a mechanization.

In the next two sections, we first explain the main concepts of our approach to the specification and verification of conformance relations between process-algebraic specifications and low-level code. Then, we present the semantic framework for low-level code, which supports the specification and verification of these conformance relations.

4.1.2 Specification and Verification of Conformance Relations

In order to overcome the so-called semantic gap introduced by the semantical distance between the different levels of abstraction, the most challenging task is to abstract from the concrete computation steps on the low-level of representation of the intermediate language. Computations that are specified as single (timed) step on the level of the high-level specification might relate to rather complex behaviors in the unstructured low-level language. Computations might be performed using, for example, loops and thus might result in

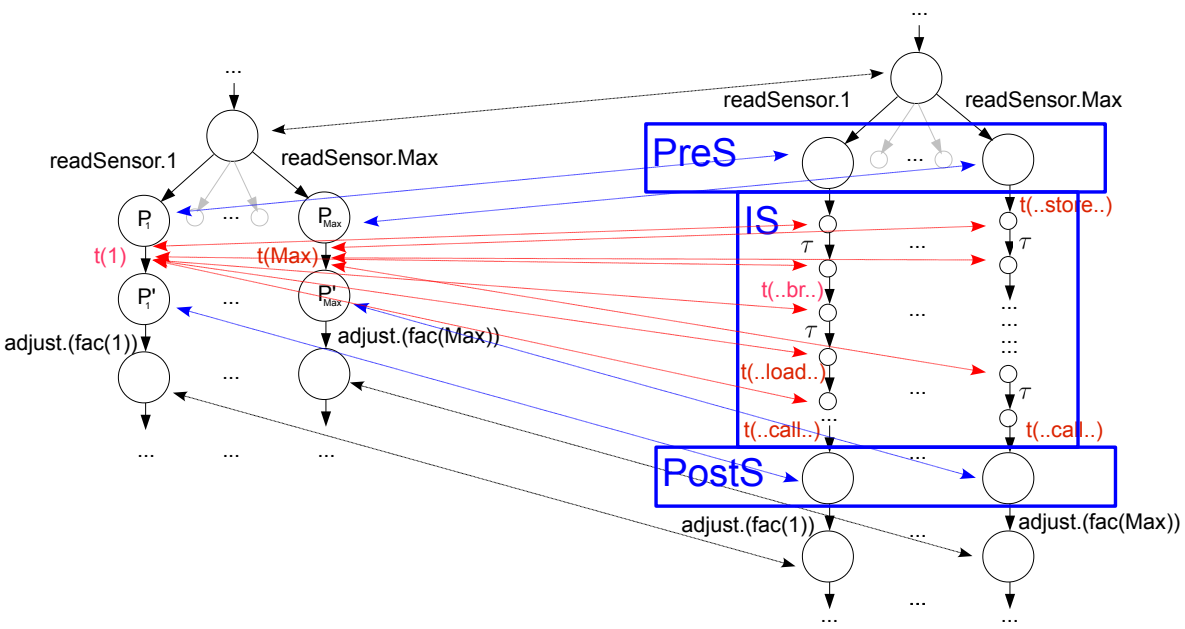

Figure 4.3: Bisimulation Relation

high numbers of individual (timed) steps. In this thesis, we therefore introduce a technique that facilitates the specification of such behaviors on a more abstract level using Hoare-triples. In this section, we explain this technique using a small example.

Informally, the notion of weak timed bisimulation is used to show that two states in an LTS have the same subsequent behavior modulo internal transitions (τ-steps). This means that for every state reachable from one of the states, there exists a state reachable from the other state from which the same behavior is possible, i.e., edges labeled with the same labels can be traversed in the same order (either time steps or event steps). Every step that is possible on the one side can be 'answered' by the other side with an equivalent step with arbitrarily many internal step before and after the event or time step.

An example is given in Figure 4.3. On the left hand side of the figure, the LTS representation of a Timed CSP process is given. After receiving a value N (ranging between 1 and Max) on channel *readSensor*, the process communicates the factorial $fac(N)$ of N on channel *adjust*. Between these two events $t(N)$ time units pass by. On the right hand side, the LTS representation of a possible low-level implementation is given. A bisimulation relation now identifies states from the right and the left hand side of the LTS and relates them. In Figure 4.3, the related states are identified by the dashed arrows. Once such a relation is specified, it needs to be proved that the property of weak timed bisimulation indeed holds for all of the identified states in the specified set. This is done by starting from the initial states and stepping through both of the LTSs, i.e., we start on the left hand side of Figure 4.3 and show for every possible step that it has a counterpart on the right hand side (preceded or followed by edges representing internal behavior) and vice versa.

While in practice this approach is feasible for Timed CSP specifications, which consist of rather few transitions, we encounter a problem when looking at the low-level side. The small nodes depicted on the right hand side of Figure 4.3 represent the state transitions within the loop that is used to calculate the factorial. Depending on the input parameter, we may find significantly more nodes compared to the Timed CSP side. Enumerating all of these states explicitly is rather unfeasible. Characterizing those chains of nodes or configurations in an abstract way on the level of the operational semantics would of course lessen the burden. However, this would lead to a rather ad-hoc proof-style and thereby complicate proof reuse. To circumvent these problems, we use a dedicated Hoare-style proof calculus (explained in the next section) for the specification and verification of weak timed bisimulations.

In Figure 4.3 on the right hand side, an assertion *PreS* describes all the low-level states that are related to the states P_i of the process algebraic specification. The assertion *PreS* is parameterized over the input parameter received in the previous transition and stored in a register. The intermediate states (*IS*) are states that are traversed while low-level instructions from a set *code* are executed. The intermediate states are only abstractly specified using the pre- and postconditions and are not enumerated explicitly. Execution moves from a state fulfilling *PreS* to a state that fulfills the postcondition *PostS*. To use such information in a bisimulation proof, it first needs to be shown that the Hoare-triple $\{PreS\}code\{PostS\}$ is indeed valid and that execution from a state fulfilling the precondition always terminates.

An interesting situation in the example are the steps from P_i to P_i' on the left hand side, which are labeled with $t(1)$, $t(2)$,...,$t(Max)$ and represent time steps. These time steps correspond to a chain of interleaved time and internal steps of the low-level implementation. For every chain on the right hand side, the sum of the individual time steps needs be equal to the timing label of the related step on the left hand side, i.e., the step labeled with $t(1)$ is answered by a series of steps with labels $t_1, ..., t_x$ and $t(1) = t_1 + ... + t_x$. To formally show that every time step can indeed be related to a series of steps within our low-level implementation, we provide an abstraction theorem. It enables us to show that every time step can be related to a step within the intermediate states without explicitly mentioning the intermediate states. Using this strategy, we can conveniently verify that every Timed CSP step has a behavioral equivalent series of low-level steps.

As mentioned above, to prove a bisimulation relation, we also need to show that every step on the low-level side can be answered from the Timed CSP side. If execution is in a state that fulfills the precondition *PreS*, we face the problem that no concrete information about the next state can be extracted from the Hoare triple, since it is a state from the set of intermediate state *IS*, which are only considered abstractly. However, the Hoare triples of our calculus contain enough information for another abstraction theorem that allows to establish the desired bisimulation property. The theorem establishes that from every intermediate state, only silent or time steps are possible and that outgoing timing edges are labeled with a time label that is a fraction of a related time

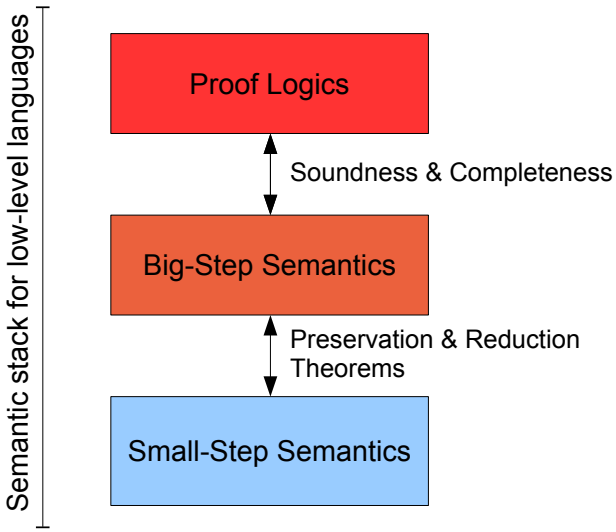

Figure 4.4: Stack of Semantics for Low-Level Languages

step on the Timed CSP side. Furthermore, from each of the intermediate states, the execution must continue to a state that fulfills the postcondition because we prove termination using our total correctness calculus. This way, we can prove that the low-level steps can be answered by Timed CSP as if every chain of state transitions between *PreS* and *PostS* was just a one step transition.

By mechanizing the abstraction theorems using Isabelle/HOL, we ensure that the correctness of the conformance relation is preserved when applying the theorems. Furthermore, automated support is provided on the level of Isabelle/HOL in order to discharge resulting proof obligations.

We observe that this technique for the specification and verification of intermediate steps in bisimulation relations between process algebraic representations and low-level representations is tightly connected to the verification techniques for low-level code. It is of particular importance that the relation between program variables specified in pre- and postconditions is correct and that, furthermore, executions from a state fulfilling the precondition always terminate in a state fulfilling the postcondition. In the following section, we explain our verification approach for low-level languages. It enables us to provide the necessary correctness conditions mentioned above.

4.1.3 Verification of Low-Level Code

The low-level language we consider is a typical one in the sense that due to conditional and unconditional branch instructions, unstructured control flow is possible. As motivated above, the specification and verification of conformance proofs is based on the small-step representation of the respective formalisms and implementation languages. We therefore provide a timed small-step semantics for a low-level implementation. To model the timing behavior

of instructions, our framework builds on the specification of timing functions. These might for example be given in terms of annotations to the low-level code. For every instruction, a certain amount of time passes by before and after the effect of the instruction can be observed. In this semantics, input and output instructions are modeled in terms of events that can be observed during execution. The observable behavior can thus be interpreted as an observable communication of the program with its environment or as an internal action (for example the update of a register). The resulting small-step semantics can be interpreted in terms of a timed labeled transition system. It therefore fits into our envisioned modular mechanization framework (see Figure 4.2) and can thus be used to relate the behavior of low-level code to Timed CSP specifications using the notion of bisimulations.

The specification of the bisimulation relation as described above is based on assertions (like *PreS* in Figure 4.3), which describe states during the execution of low-level code by focussing on certain aspects, for example the value of a variable at a certain point of execution. Furthermore, the verification of bisimulation relations is then based on possible executions of low-level code, which connect states covered by such assertions. This motivates the definition of our proof logic. It enables us to prove that, given a state described by an assertion *PreS*, after executing several low-level instructions, execution terminates in a state fulfilling *PostS*.

For our approach to the specification and verification of bisimulation relations, it is in particular important that such a proof logic is compositional, i.e., that the behavior of a combined section of low-level code can be deduced solely from its subcomponents. For low-level code, this enables us to deconstruct a program into subprograms. These subprograms can then be considered in terms of states fulfilling the precondition *PreS* before a set *code* of instructions is executed, intermediate states *IS* during execution of *code*, and states after execution of *code* that fulfill *PostS* as required by our approach to the specification and verification of bisimulations.

To obtain such a compositional proof logic for a low-level language, we follow the approach of [SU05]. There, a big-step semantics that fixes a structure on low-level code is used as an intermediate layer between the small-step semantics and the proof logic. Based on the big-step semantics, a compositional Hoare-style proof logic is defined, which closely resembles the specification style in Hoare logics for high-level languages (see 2.1).

To use this calculus for our purposes, we make two conceptual extensions. First, we introduce the possibility of non-determinism. Besides being useful for specification purposes, non-determinism can be used to model communication. Second, we provide a total correctness calculus for the non-deterministic setting. When specifying the behavior of code, as in our approach to the specification and verification of bisimulation relations, it needs to be assured that executions starting in a state fulfilling *PreS* indeed terminate in a state that fulfills *PostS* and that it does not diverge. Therefore, we prove termination between specified states using our extended proof logic. Furthermore, our big-

step semantics and proof logic is based on communication traces. For a given execution, we can therefore deduce the communications that took place.

To be able to use results obtained using our proof logic when verifying bisimulation relations, we provide preservation and reduction theorems. These theorems assure that possible behaviors of the small-step semantics are reflected by the big-step semantics. The gap between the level of the proof logics and the big-step semantics is then bridged by the soundness and completeness results for our extended proof logic. A structural overview of our semantic framework for low-level code is given in Figure 4.4. The proof logic for the verification of low-level code is the building block of the second step in our verification strategy for safety-critical embedded real-time systems (Figure 4.1). It can be used to verify the correctness of low-level code on its own, but also enables the usage of verification results established on an abstract level within conformance proofs.

In this section, we have given a detailed overview of our environment for conformance proofs between low-level code and timed process-algebraic specifications. The presented environment and the approach for the specification and verification of conformance relations also gives the theoretical background for embedded development approaches such as the one investigated in the VATES project. We give an overview of the VATES approach in the next section.

4.2 The VATES Development and Verification Approach

A source of motivation and inspiration for the verification environment presented in this thesis is the development approach investigated in the research project VATES [GBGK10]. The project investigates methods for the verification of distributed embedded real-time systems. It aims at providing an integrated development and verification approach that considers the different levels of abstraction commonly present in high assurance software development processes. One of the goals of the envisioned approach is that we want to enable developers to use established development tools, i.e., specification formalisms, programming languages and compilers.

A central problem for the design of such an approach is the complexity of commonly used high-level programming languages like, for example, C++. Advanced concepts like inheritance or templating make the definition of a formal semantics for these programming languages a quite complex undertaking. Even if the complex semantics of advanced concepts has been defined within a formal semantics, reasoning about concrete programs using such a semantics is difficult. This observation is a key motivation for the VATES development and verification approach, which is depicted in Figure 4.5.

Development starts with an abstract specification based on the process algebra CSP. CSP is a small, yet expressive, formalism targeting the specification

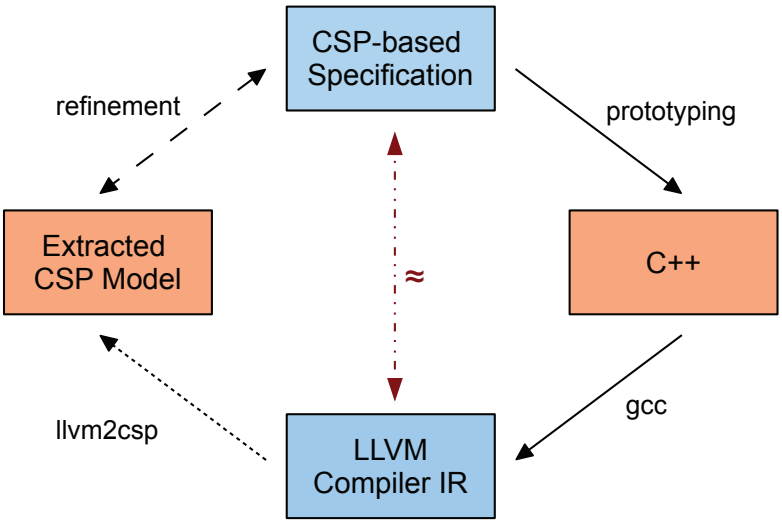

Figure 4.5: The VATES Development Approach

and analysis of concurrent systems. It is well suited for the specification and verification of properties exhibited by the embedded systems within the scope of the VATES approach. Furthermore, Timed CSP, as a conservative extension of CSP, facilitates reasoning about timing properties. [GG10] presents a mechanized proof environment for this purpose.

Using the abstract specification, a designer manually implements the system in a high-level programming language. In principle, any programming language that can be compiled to the LLVM intermediate language might be used for this purpose. However, within the VATES project, implementations in C++ are of particular interest because the main case study within the project, the small real-time operating system BOSS [MBK06], is implemented using C++. The construction of an implementation is supported by prototyping concepts [Kle11]. We assume that a programmer decorates the implementation with annotations to the source code. These annotations basically specify which points in the implementation correspond to abstract events from the CSP specifications. The annotations can be regarded as a bridge between the abstract specification and its implementation. The decorated high-level program code is then transformed to the LLVM intermediate language using the (unverified) gcc-compiler.

To establish that the resulting low-level code correctly implements the given high level specifications, the VATES approach considers two complementary approaches. In the first, the resulting low-level code along with annotations is processed by an automatic tool, called *llvm2csp* [KBG+11]. The tool extracts a low-level CSP model from the intermediate code. This low-level model can then be shown to implement the abstract specification using automatic refinement model-checking techniques. The main advantage of these techniques is that they enable a fully-automatic verification that covers all possible input scenarios. However, in practice the success of automated model-checking

approaches is often limited by their inability to deal with complex, possibly infinite state spaces. Consequently, the second approach investigated in the VATES project (and discussed in this thesis), establishes conformance using the interactive verification of bisimulation relations between different levels of abstraction [BG11]. This approach could also be used to prove the correctness of the extraction tool mentioned above [BG10].

The advantage of the VATES development approach is that it is not needed to reason about a programs behavior on the basis of the complex and rather complicated semantics of high-level languages. Instead, the proofs are established on the level of the intermediate language. Nevertheless, even though the semantics of low-level languages are less complex than the ones for high-level languages from a conceptual point of view, concrete proofs about low-level code inherit complexity due to the low-level of abstraction and the unstructured nature of such languages. Therefore, it is crucial that the verification task can be done in an automated and machine-assisted manner. In the VATES project, we achieve this by formalizing the involved formalisms and notions using the theorem prover Isabelle/HOL [NPW02] and by integrating automated refinement checkers like FDR2 and ProB into the development approach. By using the source-language independent LLVM IR as the basis for the verification, it also becomes possible to transfer the approach to development settings that use other programming languages than C++. All that needs to be done to integrate another programming language into the approach is to adjust the annotation language and to have a compiler front-end in place that transforms the respective programming language to the LLVM IR. Furthermore, using the low-level level of the intermediate language as the basis for verification, manual code optimizations (as often used in the area of embedded systems) are supported by the verification strategy. However, the correctness of the described approach builds on the transfer of properties established using abstract specifications to the implementation level. Therefore, the correctness of conformance proofs is of particular importance. The mechanized verification environment that we present in this thesis supports exactly such proofs in a machine-assisted and partly automated manner.

4.3 Summary

In this chapter, we have given an informal outline of our verification environment. To establish conformance between high-level specifications and low-level code, we use a three-step-approach. The first step is given by a concise specification of a conformance relation. In the second step, the correctness of the low-level implementation is verified using our dedicated proof calculus for timed low-level languages. The third step then establishes conformance between the low-level implementation and the abstract specification. To this end, we show that the implementation correctly implements the abstract specification by establishing the correctness of the conformance relation specified in step one. We support this task by building on the verification results from the second step. Furthermore, we have motivated how our approach for prov-

ing conformance fits into a development approach for embedded systems, like for example, the one investigated in the VATES project.

In the next chapter, we present the formalization and mechanization of a basic non-deterministic low-level language and develop a proof calculus for total correctness proofs. In the remainder of this thesis, we then further extend the proof calculus with communication primitives in a real-time setting and explain how it can be used for conformance proofs.

5 A Basic Low-Level Language in Isabelle/HOL

In the last chapter, we have described our verification environment for process-algebraic specifications and their low-level implementations. The central part of this environment is the concise representation and analysis of implementations given in their low-level representation. For the analysis of low-level code, we require an unambiguous semantics and a related proof calculus. As further requirements, we have identified that the language and the dedicated proof calculus should support proofs about non-deterministic behavior and termination in a compositional manner. Moreover, we want to be able to transfer properties established on a more abstract level to the concrete level of the small-step semantics, i.e., we aim for a semantic framework that supports the transfer of properties established using a proof logic to the level of the small-step semantics.

In this chapter, we formalize the syntax and semantics of a basic unstructured low-level language that meets the aforementioned requirements. To concentrate on the formal treatment of the desired properties, we keep the language as small as possible and abstract from timing and communication behavior at first. In Chapter 6, we then extend the semantic framework presented in this chapter with communication mechanisms and timing behavior. A compositional approach to the semantics of low-level languages and a related Hoare calculus is described in [SU05]. Our semantics and proof logic [BJ14] builds on this work and extends the approach conceptually in mainly two directions. First, we extend the simplistic low-level language with an abstract instruction *do* that can be used to model non-deterministic behavior. Second, we extend the calculus to support termination proofs, i.e., we construct a proof calculus for total correctness. The first extension enables our basic language to model a variety of programming constructs, which go way beyond the expressiveness of pure single assignments. Our extension does not impose restrictions on the kind of non-determinism and might therefore also introduce unbounded non-determinism. While the extension enables us to conveniently model potentially non-deterministic behaviors, the powerful concept comes at a cost. For the extended language, we cannot use the weakest precondition approach for the completeness proof anymore because the weakest precondition function is not continuous in this setting. As a solution, we adopt a proof strategy for

completeness used by Nipkow in [Nip02] for a structured while language and apply the strategy in the setting of unstructured low-level languages.

Our second extension is the formalization of a total correctness calculus for low-level languages. Total correctness for the original language was mentioned by the authors in a note [Saa06], but no soundness and correctness proofs were published. However, the major challenge for our approach is the fact that we deal with total correctness in the presence of non-determinism. This requires a precise characterization of the notion of termination for our extended low-level language. We show that the rules of our extended total correctness calculus, which rely on the notion of well-founded relations, are sound and complete. All proofs are formalized and mechanically checked using the theorem prover Isabelle/HOL [NPW02]. A practical contribution of the work presented in this chapter is given by a machine assisted and partly automatized verification tool for concrete low-level code, which we obtain through our mechanization. Using two examples, we show how the infrastructure of Isabelle/HOL can be used for compositional proofs about unstructured low-level code in general, and in particular how the mechanisms for handling well-founded relations provided by the theorem prover can be exploited for efficient proofs about the termination behavior of low-level code.

We start this chapter by defining the syntax of a basic low-level language. For this basic language, we then define a small-step and a big-step semantics and show that the two agree. Then, we define the rules of a partial correctness logic which we further extend to the total correctness case. We finish the chapter with two small example programs that exemplify how the presented semantic framework can be used for concrete proofs and conclude the chapter with a brief summary.

5.1 Syntax and State Definitions

In this section, we present the syntax and semantics of the simplistic low-level language for which we want to obtain a compositional big-step semantics and dedicated proof calculus. We start by giving state and syntax definitions. Then, we explain the rules of the small step semantics for the language. Building on these rules, we derive a big-step semantics for the language and establish that the two semantics are equivalent in terms of their executions. This enables the transfer of properties established on the level of the big-step semantics (obtained either by direct reasoning or using the proof calculus presented later) to the level of the small-step semantics.

We formalize the notion of state as a record with the name *state* that is parameterized over the type *'a*:

record $('a)$ *state* $= R$ $::'a$ PC $::label$

In Isabelle/HOL records offer convenient selector functions for components of the record, i.e., we can refer to the components of a given state record

s using qualified names, i.e., *R s* or *PC s*. Furthermore, components of a record can be updated in a similar fashion (as we explain in the next section). The low-level programming language that we consider operates on a store *R* of type *'a*. Using the type variable *'a* here expresses that we do not fix a concrete representation of the store. Our further definitions of instructions and semantics are polymorphic over the type *'a* as well. This implies that our formalization can later be used for concrete verification efforts by instantiating it with a convenient representation for the store, for example by using a simple function (as we do in the example later) or using more advanced concepts like records or locales for example (for a discussion on state representations see [Sch06]). The program counter *PC* points to the instruction that is to be executed next and is of type *label* (which here is a type synonym for the HOL type *nat* of natural numbers).

The low-level programming language we consider is defined using the type of instructions *('a)instr*. Instructions are parameterized over the type of the store *'a* as well:

datatype $('a)instr = do\ ('a \Rightarrow 'a\ set)\ |\ br\ label\ |\ brt\ ('a \Rightarrow bool)\ label\ label$

The instruction *do f* (of type $'a \Rightarrow 'a\ set$) applies an arbitrary (HOL) function *f* to the state. The function maps to a set of states and might thereby introduce non-determinism (if the set contains more than one state). The instructions *br* and *brt* are unconditional and conditional branching instructions and reflect the unstructured control flow of the language (in contrast to classical while languages). The conditional branch instruction depends on a predicate of type $'a \Rightarrow bool$ that maps a given state to a Boolean value. Building on these definitions, we define we define a small step semantics for the basic low-level language in the next section and explain the instructions in more detail.

5.2 Small-Step Semantics

In this section, we define a small step semantics for our low-level language. We define the transition rules of the semantics with respect to a fixed set of labeled instructions *lis*, which consists of tuples of type *(label,instr)*. The program counter points to labels and therefore to the instruction that is labeled by the respective label. We impose wellformedness conditions on this set. They state that no label refers to more than one instruction and that the set of instructions is finite. We use the *locale* concept of Isabelle/HOL to formalize this. A locale defines a named local context that allows to fix datatypes and assumptions about them. Within the context a locale, further definitions may then refer to the fixed datatypes and assumptions. Furthermore, the abstract locale definition can be instantiated concretely by defining objects and showing that these fulfill the respective assumptions. We will later use this concept when using our formal framework for proofs about concrete programs.

$$\frac{(l,\ do\ f) \in lis \qquad PC\ s = l \qquad x \in f\ (R\ s) \qquad t = s(\!|R := x,\ PC := PC\ s + 1|\!)}{(s,\ t) \in small\text{-}step}\ \text{DOF}$$

$$\frac{(l,\ br\ m) \in lis \qquad PC\ s = l \qquad t = s(\!|PC := m|\!)}{(s,\ t) \in small\text{-}step}\ \text{SMBR}$$

$$\frac{(l,\ brt\ bexp\ m\ n) \in lis \qquad PC\ s = l \qquad bexp\ (R\ s) \qquad t = s(\!|PC := m|\!)}{(s,\ t) \in small\text{-}step}\ \text{SMBRTT}$$

$$\frac{(l,\ brt\ bexp\ m\ n) \in lis \qquad PC\ s = l \qquad \neg\ bexp\ (R\ s) \qquad t = s(\!|PC := n|\!)}{(s,\ t) \in small\text{-}step}\ \text{SMBRTF}$$

Figure 5.1: Rules of the Small Step Semantics

The rules of the small-step semantics are given in Figure 5.1. They are defined within a locale context as mentioned above and refer to the fixed set of labeled instructions *lis* (and therefore to the assumptions made about it). We require labeled instruction sets to contain no two tuples with the same label. In general, a state transition from a state s to a state t is possible if the program counter in the state s ($PC\ s$) points to the label of an instruction. Further conditions have to hold for the particular instructions. If these are fulfilled, the state t is obtained from s by updating the respective components (R and PC) of the state record.

Instead of a deterministic single assignment instruction as used in [SU05], we generalize the basic low-level language to a non-deterministic setting. In the semantics of programming languages, non-determinism can be useful to provide descriptions of programs in a more abstract way. For example, if the outcome of an assignment or a section of code is not known at some design stage and certain values are possible, it might be modeled using non-determinism. In later stages of the design of a program, the respective assignment instruction can be replaced by a more concrete statement. Proofs established about the more abstract program can then be reused. Furthermore, also input instructions might be modeled as non-deterministic transitions on the level of the programming language.

The instruction *do f* formalizes the effect of an application of some HOL function f to the state s. We define it similar to the definition given in [Nip02] for a high-level language. The function *do f* yields a set of states. Therefore, for every element of this set ($x \in f\ (R\ s)$) there is a possible state t which can be reached from s. The program counter of this state t is obtained by incrementing the program counter of s by one. Note that it would have also been possible to realize an even more abstract language by allowing the *do f* instruction to also assign to the program counter. However, we choose to stress that we are dealing with a low-level language here by only allowing branching instructions to change the program counter. By allowing the function f to be defined as an arbitrary HOL function, the instruction *do f* can be used to model a variety of instructions like single assignments and binary assignments. Furthermore, skip

instructions (provided f does not change the state), abortion (if f yields the empty set) and non-deterministic instructions like non-deterministic choice and also random assignment can be modeled. The unconditional branch instruction *br m* updates the state s by setting the program counter to the value m. The last two rules are concerned with the conditional branch *brt bexp m n* . Depending on the truth value of the predicate *bexp*, evaluated with respect to the store R in state s, the instruction either updates the state by setting the program counter to m or n. Note that f and *bexp* depend only on the store R in a given state.

5.3 Big Step Semantics

In this section, we formalize a compositional big step semantics for our low-level language. To define this semantics, we follow [SU05] by imposing a structure on sets of instructions. The central idea is that any unstructured set of labeled instructions can be viewed as being constructed from subsets of labeled instructions (which do not share labeled instructions with the same label) and that, if this implicit structure is made explicit, a compositional semantics can be obtained. Given some state s, the big-step semantics then evaluates executions with respect to this structured set of instructions. We formalize this structure using the following inductively defined datatype

$$\begin{aligned}\textbf{datatype}\ ('a)\mathit{structuredCode} = \ & \mathit{none} \mid \\ & \mathit{one}\ \mathit{nat}\ ('a)\mathit{instr} \mid \\ & \mathit{seq}\ ('a)\mathit{structuredCode}\ ('a)\mathit{structuredCode}\end{aligned}$$

The constructor *none* (which has the syntax abbreviation $\emptyset$) corresponds to the empty set, while the constructor *one* (written as *label* :: *instruction*) corresponds to a set consisting of exactly one labeled instruction. The constructor *seq* (*structuredCode* $\oplus$ *structuredCode*) formalizes the intuition of structured set construction given above and is used for sequential composition in the big-step semantics. In general, a set of labeled instructions can be structured in many ways. However, the evaluations of the big-step semantics using the rules presented below are oblivious with respect to the actual fixed structure.

The rules of the big-step semantics are given in Figure 5.2. The rules for *do*, *br* and *brt* correspond to the ones from the small-step semantics. However, for the branching instructions it is required that the destination of a jump does not equal its own label. In the inference rules, the labeled instructions are now defined with respect to a piece of structured code *sc* that needs to be wellformed ($\mathit{wff}_{sc}\ sc$). Well-formedness here means that no constituent parts of a structured set of instructions share an instruction with the same label. The central part of the big-step semantics are the rules for sequential composition (*SEQ1* and *SEQ2*), and the rule *TERM* that is applicable if the program counter of a given state s does not point into the structured piece of code. The intuition behind the rules for sequential composition is the following: if execution of a piece of structured code *sc1* $\oplus$ *sc2* from a given state s starts

$$\frac{wff_{sc}\ sc \qquad sc = (l :: do\ f) \qquad (l,\ do\ f) \in lis \qquad PC\ s = l \qquad x \in f\ (R\ s) \qquad t = s(\!|R := x,\ PC := PC\ s + 1|\!)}{(s,\ sc,\ t) \in big\text{-}step}\ \text{DOF}$$

$$\frac{wff_{sc}\ sc \qquad sc = (l :: br\ m) \qquad (l,\ br\ m) \in lis \qquad PC\ s = l \qquad l \neq m \qquad t = s(\!|PC := m|\!)}{(s,\ sc,\ t) \in big\text{-}step}\ \text{BR}$$

$$\frac{wff_{sc}\ sc \qquad sc = (l :: brt\ bexp\ m\ n) \qquad (l,\ brt\ bexp\ m\ n) \in lis \qquad PC\ s = l \qquad bexp\ (R\ s) \qquad l \neq m \qquad t = s(\!|PC := m|\!)}{(s,\ sc,\ t) \in big\text{-}step}\ \text{BRTT}$$

$$\frac{wff_{sc}\ sc \qquad sc = (PC\ s :: brt\ bexp\ m\ n) \qquad (l,\ brt\ bexp\ m\ n) \in lis \qquad PC\ s = l \qquad \neg\ bexp\ (R\ s) \qquad l \neq n \qquad t = s(\!|PC := n|\!)}{(s,\ sc,\ t) \in big\text{-}step}\ \text{BRTF}$$

$$\frac{sc = (sc1 \oplus sc2) \qquad PC\ s \in_{sc} sc1 \qquad wff_{sc}\ sc \qquad (s,\ sc1,\ s1) \in big\text{-}step \qquad (s1,\ sc1 \oplus sc2,\ t) \in big\text{-}step}{(s,\ sc,\ t) \in big\text{-}step}\ \text{SEQ1}$$

$$\frac{sc = (sc1 \oplus sc2) \qquad PC\ s \in_{sc} sc2 \qquad wff_{sc}\ sc \qquad (s,\ sc2,\ s1) \in big\text{-}step \qquad (s1,\ sc1 \oplus sc2,\ t) \in big\text{-}step}{(s,\ sc,\ t) \in big\text{-}step}\ \text{SEQ2}$$

$$\frac{\neg\ (PC\ s \in_{sc} sc) \qquad wff_{sc}\ sc}{(s,\ sc,\ s) \in big\text{-}step}\ \text{TERM}$$

Figure 5.2: Rules of the Bigstep Semantics

in the first part of the structured piece of code (because $PC\ s$ points to a label within *sc1*), then the code *sc1* is executed from this state. After this, a state *s1* with a program counter outside of *sc1* is obtained. From this state both parts of the original structured code ($sc1 \oplus sc2$) are executed. The reason for considering both parts of the composition is that from *sc2* there might be a jump back into *sc1*. This way of executing the two pieces of code is done until a state is reached where the program pointer is outside of $sc1 \oplus sc2$.

The big-step semantics corresponds to the small-step semantics, i.e., for a given execution in the big-step semantics there exists a corresponding execution in the small-step semantics and for stuck executions of the small-step semantics (i.e., if the program pointer does not point to a labeled instruction anymore) there exists a corresponding big-step execution. These simulation and reduction theorems enable the transfer of properties established on the level of the big-step semantics to the small-semantics. Thereby, the big-step semantics serves as a connection layer between the unstructured layer of the small-step semantics and the structured and more abstract layer of the proof calculus, which we present in the upcoming section.

5.4 Hoare-Logics for Low-Level Languages

In this section, we formalize the rules of our Hoare logic for non-deterministic low-level languages. As motivated in the introduction, we have extended our low-level language with non-determinism. The main obstacle in order to make the proof calculus compatible/working with this extension is the completeness proof. Since the weakest precondition approach does not work for non-deterministic languages, we employ the notion of the most general triple to mechanically prove that the rules of our calculus are indeed complete (for a discussion on relative completeness we refer to [Nip02]). Furthermore, we extend the calculus to handle auxiliary variables explicitly. In the next subsection, we start by presenting our calculus for partial correctness and establish soundness and completeness. In the subsequent section, we extend the rules of the partial correctness calculus to account for termination and explain how we establish soundness and completeness in this setting.

As common in formalizations of Hoare calculi in theorem provers, in our formalization, assertions are formalized using the extensional approach, i.e., we only fix the type of assertions but do not define an assertion language explicitly. We model auxiliary variables by formalizing an auxiliary state. Auxiliary variables are used to freeze the values of variables before and after execution of some code fragment. Using auxiliary variables, it is thereby possible to refer to variable values from states fulfilling a precondition in a postcondition. The type of assertions is formalized as:

type-synonym $('aux,'a)ass = 'aux \Rightarrow ('a)state \Rightarrow bool$

An assertion is applied to a state and yields a Boolean value that reflects whether the assertion is fulfilled in the given state or not. The type of assertions is defined as being polymorphic with respect to the type of auxiliary state *'aux* and the type of state *'a* . Thereby, the auxiliary state can be instantiated conveniently when using the calculus. For example, in the completeness proof we instantiate the auxiliary state with the type of state.

5.4.1 Partial Correctness

The rules of the partial correctness calculus are depicted in Figure 5.3. The first three rules are similar to the standard Hoare rules of structured languages, i.e., the precondition is defined using the assertion q that defines the postcondition. This assertion q is evaluated in a state that is substituted according to the effects of the instruction that the rule refers to (see Section 2.1). Furthermore, the assertion q is evaluated with respect to the auxiliary state aux. This allows to specify an auxiliary state which can be used to refer to variables from the precondition in the postcondition. The definition of validity given below ensures that both the precondition and the postcondition are evaluated with respect to the same auxiliary state aux. Regarding the assertions there is an important difference compared to the assertions used in Hoare calculi for high-

$$\left\{\begin{array}{l}\lambda\ aux\ s.PC\ s = l \wedge \\ \qquad \forall x{\in}f\ (R\ s).\ q\ aux\ (s(\!|R := x,\ PC := PC\ s + 1|\!)) \\ \quad \vee\ PC\ s \neq l \wedge q\ aux\ s\end{array}\right\} l :: do\ f\ \{q\}\ \text{DOF}$$

$$\left\{\begin{array}{l}\lambda\ aux\ s.PC\ s = l \wedge \\ \qquad (q\ aux\ (s(\!|PC := m|\!)) \vee m = l) \\ \quad \vee\ PC\ s \neq l \wedge q\ aux\ s\end{array}\right\} l :: br\ m\ \{q\}\ \text{BR}$$

$$\left\{\begin{array}{l}\lambda\ aux\ s.PC\ s = l \wedge \\ \qquad (b\ (R\ s) \wedge (q\ aux\ (s(\!|PC := m|\!)) \vee m = l) \vee \\ \qquad \neg\ b\ (R\ s) \wedge (q\ aux\ (s(\!|PC := n|\!)) \vee n = l)) \\ \quad \vee\ PC\ s \neq l \wedge q\ aux\ s\end{array}\right\} l :: brt\ b\ m\ n\ \{q\}\ \text{BRTF}$$

$$\frac{\begin{array}{c}\vdash_p \{\lambda aux\ s.\ (PC\ s \in_{sc} sc1) \wedge i\ aux\ s\}\ sc1\ \{i\} \\ \vdash_p \{\lambda aux\ s.\ (PC\ s \in_{sc} sc2) \wedge i\ aux\ s\}\ sc2\ \{i\}\end{array}}{\vdash_p \{i\}\ sc1 \oplus sc2\ \{\lambda aux\ s.\ \neg\ (PC\ s \in_{sc} sc1) \wedge \neg\ (PC\ s \in_{sc} sc2) \wedge i\ aux\ s\}}\text{SEQ}$$

$$\frac{\begin{array}{c}\vdash_p \{p'\}\ sc\ \{q'\} \\ \forall\ s\ t.\ (\forall\ aux.\ p'\ aux\ s \longrightarrow q'\ aux\ t) \longrightarrow (\forall\ aux.\ p\ aux\ s \longrightarrow q\ aux\ t)\end{array}}{\vdash_p \{p\}\ sc\ \{q\}}\text{CONS}$$

$$\vdash_p \{p\}\ \emptyset\ \{p\}\text{NONE}$$

Figure 5.3: Rules of the Partial Correctness Calculus

level languages. We deal with low-level code here that is multi-entry and multi-exit, which means that program code can potentially be executed starting at any labeled instruction in it. Our assertions therefore explicitly refer to the value of the instruction counter. Therefore, specifications are also valid if the instruction pointer in a particular state does not point to the actual instruction that the rule refers to, but fulfills the postcondition without executing the code. The rule for the instruction *do f* specifies that q needs to hold for every state s from the set obtained from the application of f to the current store $(R\ s)$. The rule for *br m* evaluates the assertion q in a state where the program counter is set to m. The disjunct $m = l$ is used to cope with branching instruction that jump to their own label. In this case, the precondition evaluates to true and any postcondition is possible. The rule for conditional branches is defined similar to the one for unconditional branches but depends on a Boolean condition b. The postcondition q needs to hold for a state in which the program counter points to m if the branch condition b evaluates to true and for a state where the program counter points to n if the branch condition b evaluates to false with respect to the current store $R\ s$. The rule for sequential composition resembles both the rule for sequential composition and for loops known from standard calculi for structured languages. An invariant is needed here for the three states involved in a sequential composition: the state before the first part of

the structured code (*sc1*), the intermediate state reached after executing *sc1* and before executing *sc2* and the state after executing *sc2*. The invariant needs to be strong enough to cope with possible jumps between the two pieces of structured code. Note that the invariant only needs to refer to the program counter values from which the respective pieces of structured code might be entered or left with respect to the execution of the subpieces. All other labels that might be visited for example by executing a sequence of straight line code within one of the pieces of code involved in the sequential composition need not to be specified in the invariant. The intuition behind the rule is that if the structured code $sc1 \oplus sc2$ is executed from a state with a program counter specified in the invariant, execution ends in a state that again fulfills the invariant but in which the program counter is outside of $sc1 \oplus sc2$. The rule of consequence is quite different from the one in [SU05] because it is also used to adapt the auxiliary state. The classical rules for strengthening the precondition and weakening the postcondition can be derived from this one.

A specification of the partial correctness calculus is valid iff from every state that fulfills the precondition p, execution of the instructions from *code* ends in a state that fulfills the postcondition q (if execution of *code* terminates). This intuition yields the following definition:

Definition 10 (Validity for partial correctness)
$\models_p \{p\}\ code\ \{q\} \longleftrightarrow \forall\, aux\ s\ t.\ p\ aux\ s \wedge (s,\ code,\ t) \in big\text{-}step \longrightarrow q\ aux\ t$

The rules of Figure 5.3 are sound with respect to the rules of the operational semantics. This is established by rule induction over the rules of the proof system. Here, we show in detail how such a proof is realized in Isabelle/HOL. We explain the proof commands that are used to discharge the subgoals appearing within the proof.

Theorem 1 (Soundness for partial correctness)
$\vdash_p \{p\}\ code\ \{q\} \longrightarrow \models_p \{p\}\ code\ \{q\}$

As a first step, we expand the definition of validity given above using the command `unfold`.

```
apply(clarsimp, unfold valid_def)
```

After unfolding the definition of validity the following subgoal is obtained:

$\vdash_p \{p\}\ code\ \{q\} \Longrightarrow \forall\, aux\ s\ t.\ p\ aux\ s \wedge (s,\ code,\ t) \in big\text{-}step \longrightarrow q\ aux\ t$

We perform induction over the rules of the proof calculus. This is achieved by using the automatically generated induction principle `vspec.induct` as an elemination rule. The name *vspec* refers to our inductively defined proof rules.

```
apply(erule vspec.induct)
```

After applying the command, we obtain six subgoals. The first one results from the proof rule for the instruction `do f`:

$PC\ s = l \wedge (\forall x{\in}f\ (R\ s).\ q\ aux\ (s(\!|R := x,\ PC := Suc\ (PC\ s)|\!))) \vee$
$PC\ s \neq l \wedge q\ aux\ s \Longrightarrow$
$(s,\ l :: do\ f,\ t) \in big\text{-}step \Longrightarrow q\ aux\ t$

This subgoal can be automatically discharged using the following command:

```
apply(clarsimp, erule big_step.cases, clarsimp+)
```

Here, we first simplify the subgoal and then perform a case distinction of the rules of the big-step semantics by applying the theorem `big_step.cases` (automatically generated by Isabelle for the inductive definition of the relation `big_step`) as an elimination rule.

The same proof commands can be used to discharge the subgoal resulting from the rule for the instruction `br`:

$PC\ s = l \wedge (q\ aux\ (s(\!|PC := m|\!)) \vee m = l) \vee PC\ s \neq l \wedge q\ aux\ s \Longrightarrow$
$(s,\ l :: br\ m,\ t) \in big\text{-}step \Longrightarrow q\ aux\ t$

The subgoal corresponding to the instruction `brt` is the following.

$PC\ s = l \wedge (b\ (R\ s) \wedge (q\ aux\ (s(\!|PC := m|\!)) \vee m = l) \vee$
$\quad \neg\ b\ (R\ s) \wedge (q\ aux\ (s(\!|PC := n|\!)) \vee n = l)) \vee$
$PC\ s \neq l \wedge q\ aux\ s \Longrightarrow$
$(s,\ l :: brt\ b\ m\ n,\ t) \in big\text{-}step \Longrightarrow q\ aux\ t$

We use the same proof strategy as for the instruction `br` here. However, event though simplification using the proof command `clarsimp` is not sufficient here, Isabelle/HOL is still able to discharge the subgoal automatically using the automated reasoning method `fast`. This leads to the following series of proof commands:

```
apply(clarsimp, erule big_step.cases, fast+, clarsimp)
```

The following subgoal results from the rule for sequential composition:

$\vdash_p \{\lambda aux\ s.\ (PC\ s \in_{sc} sc1) \wedge i\ aux\ s\}\ sc1\ \{i\} \Longrightarrow$
$\forall\ aux\ s\ t.\ ((PC\ s \in_{sc} sc1) \wedge i\ aux\ s) \wedge (s,\ sc1,\ t) \in big\text{-}step \longrightarrow i\ aux\ t \Longrightarrow$
$\vdash_p \{\lambda aux\ s.\ (PC\ s \in_{sc} sc2) \wedge i\ aux\ s\}\ sc2\ \{i\} \Longrightarrow$
$\forall\ aux\ s\ t.\ ((PC\ s \in_{sc} sc2) \wedge i\ aux\ s) \wedge (s,\ sc2,\ t) \in big\text{-}step \longrightarrow i\ aux\ t \Longrightarrow$
$i\ aux\ s \Longrightarrow (s,\ sc1 \oplus sc2,\ t) \in big\text{-}step \Longrightarrow$
$\neg\ (PC\ t \in_{sc} sc1) \wedge \neg\ (PC\ t \in_{sc} sc2) \wedge i\ aux\ t$

To prove this subgoal we use an auxiliary lemma. We choose to do so because we need to perform an induction over the executions of the big-step semantics. Using the following lemma called *seqInduct*, the Isabelle system is able to perform the unification correctly when performing induction over the rules of the big-step semantics.

$(p,\ c,\ q) \in big\text{-}step \Longrightarrow$
$\forall\ sc1\ invariant\ sc2\ aux\ s\ t.\ c = (sc1 \oplus sc2) \wedge p = s \wedge q = t \wedge$
$\vdash_p \{\lambda aux\ s.\ (PC\ s \in_{sc} sc1) \wedge invariant\ aux\ s\}\ sc1\ \{invariant\} \wedge$
$(\forall\ aux\ s\ t.\ ((PC\ s \in_{sc} sc1) \wedge invariant\ aux\ s) \wedge (s,\ sc1,\ t) \in big\text{-}step \longrightarrow$
$invariant\ aux\ t) \wedge$

$\vdash_p \{\lambda aux\ s.\ (PC\ s \in_{sc} sc2) \wedge invariant\ aux\ s\}\ sc2\ \{invariant\} \wedge$
$(\forall\ aux\ s\ t.\ ((PC\ s \in_{sc} sc2) \wedge invariant\ aux\ s) \wedge (s,\ sc2,\ t) \in big\text{-}step \longrightarrow$
$invariant\ aux\ t) \wedge$
$invariant\ aux\ s$
$\longrightarrow \neg\ (PC\ t \in_{sc} sc1) \wedge \neg\ (PC\ t \in_{sc} sc2) \wedge invariant\ aux\ t$

The lemma states that given a big-step execution of a sequentially composed piece of code, execution terminates in a state t where the program pointer does not point into the composed pieces of code anymore. Furthermore, the *invariant* holds in t if execution starts in a state fulfilling the *invariant* and the given partial correctness specifications hold for the individual pieces of code involved in the composition. We use the lemma to prove the original subgoal by adding it to the assumptions of the original subgoal using the command `insert` and instantiating the universally quantified meta variables. These result from the fact that the variables p, c and q are not universally quantified in our lemma and are therefore interpreted as meta variables.

```
apply(clarify,insert seqInduct)
apply(erule_tac x="s" in meta_allE)
apply(erule_tac x="seq sc1 sc2" in meta_allE)
apply(erule_tac x="t" in meta_allE)
```

To use the lemma in our soundness proof, we need to establish that the assumption about the big-step execution holds in the current proof context. This is achieved using the following proof command.

```
apply(drule meta_mp, assumption)
```

After instantiating the universally quantified variables of our lemma *seqInduct* appropriately, the subgoal is proved by automatic means using the proof method `fast`:

```
apply(erule_tac x="sc1" in allE)
apply(erule_tac x="i" in allE)
apply(erule_tac x="sc2" in allE)
apply(rotate_tac 4, erule_tac x="aux" in allE)
apply(rotate_tac 6, erule_tac x="s" in allE)
apply(rotate_tac 6, erule_tac x="t" in allE)
apply(fast)
```

Following the application of the command `clarsimp`, the following subgoal remains for the rule of consequence:

$\vdash_p \{p'\}\ sc\ \{q'\} \Longrightarrow$
$\forall\ aux\ s\ t.\ p'\ aux\ s \wedge (s,\ sc,\ t) \in big\text{-}step \longrightarrow q'\ aux\ t \Longrightarrow$
$\forall\ s\ t.\ (\forall\ aux.\ p'\ aux\ s \longrightarrow q'\ aux\ t) \longrightarrow (\forall\ aux.\ p\ aux\ s \longrightarrow q\ aux\ t) \Longrightarrow$
$p\ aux\ s \Longrightarrow (s,\ sc,\ t) \in big\text{-}step \Longrightarrow q\ aux\ t$

Since the rule involves no information about concrete executions of the big step semantics, we can prove the subgoal by using the automatic proof method `fast`.

For the empty piece of structured code, we obtain the following subgoal:

$p\ aux\ s \Longrightarrow (s,\ \emptyset,\ t) \in big\text{-}step \Longrightarrow p\ aux\ t$

Again this subgoal can be proved by using the respective definition within the big-step semantics using the following proof command:

```
apply(clarsimp, erule big_step.cases, clarsimp+)
```

After proving the last subgoal, the soundness proof is finished using the command `done`.

The rules for the partial correctness calculus yield a complete proof system with respect to well-formed structured code (see above for well-formedness).

Theorem 2 (Completeness for partial correctness)
If $wff_{sc}\ code$ ***and*** $\cup_{sc}\ code \subseteq lis$ ***then***
$\models_p \{P\}\ code\ \{Q\} \longrightarrow \vdash_p \{P\}\ code\ \{Q\}$

The proof uses the most general triple approach. In the next section, we explain the completeness proof for total correctness in more detail. The proof for the total correctness case subsumes the one for partial correctness. Therefore, we skip the details here.

In this section, we have presented a partial correctness calculus for our basic non-deterministic low-level language. As motivated in the introduction to our proof environment for conformance proofs, we require a total correctness calculus for our approach. In the next section, we therefore extend the partial correctness calculus to total correctness and enable proofs about the termination behavior of low-level code.

5.4.2 Total Correctness

In this section, we describe our adapted proof calculus for the total correctness case. The extension is similar to the way total correctness is realized in proof calculi for structured programming languages in the sense that the specification of a variant is required in the rule for sequential composition. Informally, for high level languages such a variant ensures that after every loop iteration a value of the state (or a combination of values) is decreased. If a lower bound is known for the values of the respective variable, the loop must terminate at some point. More precisely, a well-founded relation defined on the state of the programming language is required, which formalizes that the state is decreased with every loop iteration. Recall that the rule of sequential composition in our proof system for low-level code corresponds to both the rules for sequential composition and loops in Hoare calculi for high-level programming languages. For the rule of sequential composition in our proof calculus, we can use the same approach as for high-level languages. The rules of the total correctness calculus are given in Figure 5.4.

Compared to the partial correctness rules, the rule for *do f* now further requires that the set of states yield by *f* is not empty. For the branching instructions, it is required that they do not branch to their own label. Regarding

$$\left\langle \begin{array}{l} \lambda\ aux\ s.PC\ s = l \wedge f\ (R\ s) \neq \emptyset \wedge \\ \qquad \forall x \in f\ (R\ s).\ q\ aux\ (s(\!|R := x,\ PC := PC\ s + 1|\!)) \\ \quad \vee\ PC\ s \neq l \wedge q\ aux\ s \end{array} \right\rangle l :: do\ f \langle q \rangle \ \text{HDOF}$$

$$\left\langle \begin{array}{l} \lambda\ aux\ s.PC\ s = l \wedge \\ \qquad q\ aux\ (s(\!|PC := m|\!)) \wedge m \neq l \\ \quad \vee\ PC\ s \neq l \wedge q\ aux\ s \end{array} \right\rangle l :: br\ m \langle q \rangle \ \text{HBR}$$

$$\left\langle \begin{array}{l} \lambda\ aux\ s.PC\ s = l \wedge \\ \qquad (b\ (R\ s) \wedge q\ aux\ (s(\!|PC := m|\!)) \wedge m \neq l \vee \\ \qquad \neg\ b\ (R\ s) \wedge q\ aux\ (s(\!|PC := n|\!)) \wedge n \neq l) \\ \quad \vee\ PC\ s \neq l \wedge q\ aux\ s \end{array} \right\rangle l :: brt\ b\ m\ n \langle q \rangle \ \text{HBRTF}$$

$$\frac{\begin{array}{c} wf\ r \\ \forall s'.\ \vdash_t \langle \lambda aux\ s.\ (PC\ s \in_{sc} sc1) \wedge i\ aux\ s \wedge s = s' \rangle\ sc1\ \langle \lambda aux\ s.\ i\ aux\ s \wedge (s,\ s') \in r \rangle \\ \forall s'.\ \vdash_t \langle \lambda aux\ s.\ (PC\ s \in_{sc} sc2) \wedge i\ aux\ s \wedge s = s' \rangle\ sc2\ \langle \lambda aux\ s.\ i\ aux\ s \wedge (s,\ s') \in r \rangle \end{array}}{\vdash_t \langle i \rangle\ sc1 \oplus sc2\ \langle \lambda aux\ s.\ \neg\ (PC\ s \in_{sc} sc1) \wedge \neg\ (PC\ s \in_{sc} sc2) \wedge i\ aux\ s \rangle}\ \text{HSEQ}$$

$$\frac{\vdash_t \langle p' \rangle\ sc\ \langle q' \rangle \qquad \begin{array}{c} \forall s\ t.\ (\forall\ aux.\ p'\ aux\ s \longrightarrow q'\ aux\ t) \longrightarrow (\forall\ aux.\ p\ aux\ s \longrightarrow q\ aux\ t) \\ \forall s.\ (\exists\ aux.\ p\ aux\ s) \longrightarrow (\exists\ aux.\ p'\ aux\ s) \end{array}}{\vdash_t \langle p \rangle\ sc\ \langle q \rangle}\ \text{HCONS}$$

$$\vdash_t \langle p \rangle\ \emptyset\ \langle p \rangle \text{HNONE}$$

Figure 5.4: Rules of the Total Correctness Calculus

sequential composition, a well-founded relation needs to be provided (condition *wf r*). A binary relation is well-founded iff it does not allow for infinitely descending chains. For example, the relation less-than on the natural numbers is a well-founded relation. Given such a relation, it needs to be established for both of the sub-specifications that the state is decreased with respect to this relation. This is formalized by the term $s = s'$ in the precondition and the term $(s, s') \in r$ in the postcondition of the assumptions for the rule of sequential composition. The rule of consequence is similar to the one for partial correctness, but adds a conjunct for strengthening the auxiliary state in the precondition.This is required to achieve adaption completeness [Sch97, Old83]. Informally, adaption completeness expresses that the auxiliary state can always be adjusted as required in order to express arbitrary valid specifications. This is of particular importance when extending a proof system in order to handle recursive function calls, for example. The treatment of recursion can be significantly simplified when using auxiliary variables [Sch97].

Validity of a specification derived using the inference rules of our total correctness calculus is defined similarly to validity in the context of partial correctness. Additionally, we require that if a state s fulfills the precondition p, the execution of the respective piece of structured code *code* terminates for

all executions of *code* starting in s. This is formalized using the termination predicate $\downarrow$ defined by the rules in Figure 5.5 and reflected by the last conjunct of the term in the following definition:

Definition 11 (Validity for total correctness)
$\models_t \langle p \rangle\ code\ \langle q \rangle \longleftrightarrow$
$\quad \forall\, aux\ s\ t.\ p\ aux\ s \wedge (s,\ code,\ t) \in big\text{-}step \longrightarrow q\ aux\ t\ \wedge$
$\quad \forall\, aux\ s.\ p\ aux\ s \longrightarrow code \downarrow s$

A specification is then valid (abbreviated $\models_t \langle p \rangle\ code\ \langle q \rangle$) iff for all states that fulfill the precondition p and from which *code* is executed, the postcondition q holds and the execution of *code* terminates. The axiomatization of termination via the predicate $\downarrow$ is necessary, since in the presence of non-determinism the existence of a terminating execution is not sufficient to claim that all executions from a certain state terminate. The intuition behind the formalized rules is as follows. As mentioned above, the instruction *do f* can be used to model an instruction that blocks execution given that the function f yields the empty set. Therefore, the termination rule for *do f* requires the set of states yield by f to be non-empty. For the branching instructions, it is obvious that they terminate if they do not jump to their own label. All instructions terminate if they are evaluated in a state in which the instruction pointer does not point to the instruction. The empty structured instruction ($\emptyset$) terminates for any state since it contains no labeled instructions that the program counter in a given state might point to. The most interesting case is the sequential composition. Here, termination behavior is defined inductively. A sequential composition terminates from a given state s, if execution of the first part of the structured code of the composition from s terminates in a state t, and then the sequential composition terminates if executed from t or vice versa. If the program pointer in s does not point into the sequential composition it terminates as well.

Theorem 3 (Soundness for total correctness)
$\vdash_t \langle p \rangle\ code\ \langle q \rangle \longrightarrow \models_t \langle p \rangle\ code\ \langle q \rangle$

We prove soundness by invoking induction on the derivation rules of the Hoare logic. The proof is realized similarly to the proof for the partial correctness case. For the basic instructions it is sufficient to unfold the definitions of the respective rule from the big-step semantics and the termination predicate $\downarrow$. The most interesting case is the sequential composition. The proof requires induction over the big-step semantics. To establish that the termination predicate holds for a sequentially composed piece of code, we invoke well-founded induction using the provided well-founded relation r.

We prove completeness using the most general triple approach [Gor75, Nip02] because the weakest precondition approach does not work in the presence of unbounded non-determinism. Formally, the most general triple is defined as follows:

Definition 12 (Most general triple)
$mgt\ code \equiv (\lambda z\ s.\ z = s \wedge code \downarrow s,\ code,\ \lambda z\ t.\ (z,\ code,\ t) \in big\text{-}step)$

$$\frac{f\ (R\ s) \neq \emptyset \vee PC\ s \neq l}{(l :: do\ f) \downarrow s}\text{DOF} \qquad \frac{m \neq l \vee PC\ s \neq l}{(l :: br\ m) \downarrow s}\text{BR}$$

$$\frac{b\ (R\ s) \wedge m \neq l \vee \neg\ b\ (R\ s) \wedge n \neq l \vee PC\ s \neq l}{(l :: brt\ b\ m\ n) \downarrow s}\text{BRTF}$$

$$\frac{\begin{array}{c} PC\ s \in_{sc} (sc\text{-}1 \oplus sc\text{-}2) \\ ((PC\ s \in_{sc} sc\text{-}1) \wedge sc\text{-}1 \downarrow s \wedge (\forall t.\ (s, sc\text{-}1, t) \in big\text{-}step \longrightarrow (sc\text{-}1 \oplus sc\text{-}2) \downarrow t) \vee \\ (PC\ s \in_{sc} sc\text{-}2) \wedge sc\text{-}2 \downarrow s \wedge (\forall t.\ (s, sc\text{-}2, t) \in big\text{-}step \longrightarrow (sc\text{-}1 \oplus sc\text{-}2) \downarrow t)) \end{array}}{(sc\text{-}1 \oplus sc\text{-}2) \downarrow s}\text{SEQ}$$

$$\frac{\neg\ (PC\ s \in_{sc} (sc\text{-}1 \oplus sc\text{-}2))}{(sc\text{-}1 \oplus sc\text{-}2) \downarrow s}\text{SEQ2}$$

$$\emptyset \downarrow s\text{NONE}$$

Figure 5.5: Definition of the Termination Predicate

The intuition behind the most general triple is that it reflects all possible executions from a state that fulfills the precondition, by 'freezing' this state using the auxiliary state z and then claiming in the postcondition that it holds for all states t which can be reached via the big-step semantics. Since we deal with total correctness, it is also claimed that execution from a state that fulfills the precondition needs to terminate (using the termination predicate $\downarrow$). If it can be established that the rules of the Hoare calculus are sufficient to prove the most general triple it follows that the calculus is complete:

Lemma 1 (Most general triple implies completeness)
If $wff_{sc}\ code$ ***and*** $\vdash_t \langle fst\ (mgt\ code)\rangle\ code\ \langle snd\ (snd\ (mgt\ code))\rangle$ ***then*** $\models_t \langle P\rangle\ code\ \langle Q\rangle \longrightarrow \vdash_t \langle P\rangle\ code\ \langle Q\rangle$

The most general triple is defined with respect to a piece of structured code *code* and consists of three parts. We therefore refer to the first part (via *fst*) and the third part (via *snd(snd(x))*) using Isabelle/HOL's selector functions for pairs in the following lemma, which establishes that the rules of our proof system are sufficient to derive the most general triple.

Lemma 2 (Derivation of the most general triple)
If $\cup_{sc}\ code \subseteq lis$ ***and*** $wff_{sc}\ code$ ***then*** $\vdash_t \langle fst\ (mgt\ code)\rangle\ code\ \langle snd\ (snd\ (mgt\ code))\rangle$

The proof is done by induction on the structured form of *code* (which needs to be well-formed). We show the case for the instruction *do f* here. The other cases (apart from sequential composition) are similar. We need to show that the following triple holds:

$$\vdash_t \langle \lambda z\ s.\ z = s \wedge (l :: do\ f) \downarrow s \rangle\ l :: do\ f\ \langle \lambda z\ t.\ (z,\ l :: do\ f,\ t) \in \mathit{big\text{-}step} \rangle$$

Using the rule of consequence we strengthen the precondition to obtain the following triple:

$$\begin{array}{l}
\vdash_t \langle \lambda aux\ s.\ PC\ s = l\ \wedge f\ (R\ s) \neq \emptyset \wedge \\
\qquad \forall x{\in}f\ (R\ s).\ (aux,\ l :: do\ f,\ s(\!|R := x,\ PC := PC\ s + 1|\!)) \in \mathit{big\text{-}step} \\
\qquad\quad \vee PC\ s \neq l \wedge (aux,\ l :: do\ f,\ s) \in \mathit{big\text{-}step} \rangle \\
\qquad l :: do\ f \\
\quad \langle \lambda z\ t.\ (z,\ l :: do\ f,\ t) \in \mathit{big\text{-}step} \rangle
\end{array}$$

Using the proof rule for the instruction *do f* we want to show that this triple can be derived in our proof system. To apply the rule, we first need to apply the rule of consequence. The application of the rule of consequence requires to show that the following holds:

$$\begin{array}{l}
\textbf{If}\ (l :: do\ f) \downarrow s\ \textbf{and}\ PC\ s = l \vee (s,\ l :: do\ f,\ s) \notin \mathit{big\text{-}step}\ \textbf{then} \\
\quad PC\ s = l \wedge f\ (R\ s) \neq \emptyset \wedge \\
\quad \forall x{\in}f\ (R\ s).\ (s,\ l :: do\ f,\ s(\!|R := x,\ PC := PC\ s + 1|\!)) \in \mathit{big\text{-}step}
\end{array}$$

The definition of termination in the first assumption ($(l :: do\ f) \downarrow s$) implies that the function f should not yield the empty set ($f\ (R\ s) \neq \emptyset$). This already shows the second conjunction of our proof goal. The second assumption in this proof obligation states that either the program counter points to instruction *do f* at label l or there exists no transition from s to itself through the instruction *do f* at label l. It is easy to show that the latter can only be the case if the instruction pointer indeed points to l. We therefore need to show:

$$\forall x{\in}f\ (R\ s).\ (s,\ PC\ s :: do\ f,\ s(\!|R := x,\ PC := PC\ s + 1|\!)) \in \mathit{big\text{-}step}$$

which follows from the big-step rule for *do f* since the program counter of s points to the label of the *do f* instruction.

Proofs for the other instructions are similar. For sequential composition, we have the assumption that the following triples hold for the individual parts of the sequential composition:

$$\vdash_t \langle \lambda z\ s.\ z = s \wedge \mathit{sc1} \downarrow s \rangle\ \mathit{sc1}\ \langle \lambda z\ t.\ (z,\ \mathit{sc1},\ t) \in \mathit{big\text{-}step} \rangle \qquad (1)$$
$$\vdash_t \langle \lambda z\ s.\ z = s \wedge \mathit{sc2} \downarrow s \rangle\ \mathit{sc2}\ \langle \lambda z\ t.\ (z,\ \mathit{sc2},\ t) \in \mathit{big\text{-}step} \rangle \qquad (2)$$

and we have to establish the following triple:

$$\vdash_t \langle\ \lambda z\ s.\ z = s \wedge (sc1 \oplus sc2) \downarrow s\rangle \qquad (3)$$
$$sc1 \oplus sc2$$
$$\langle\lambda z\ t.\ (z,\ sc1 \oplus sc2,\ t) \in \mathit{big\text{-}step}\rangle$$

To apply the appropriate proof rule for sequential composition, we need to find a suitable invariant and provide a well-founded relation for the composition. Both are based on the observation that the execution of the sequential composition might alternate between the two pieces of structured code involved in the sequential composition. Therefore, we define the invariant based on the transitive closure of the set of state-pairs (u, v) where state v can be reached from state u through *sc1* or *sc2*. The invariant then describes for a tuple of states (z, t) all the states z visited after and between executing *sc1* or *sc2* in an alternating way. Note that states visited 'within' the execution of either of the pieces of code are not captured. Furthermore, the invariant requires the alternating executions to terminate. Using the rule of consequence, we strengthen the precondition of the desired specification triple (3) to the following assertion, which exactly formalizes the invariant described above using the termination requirement:

$$\begin{aligned}\langle\lambda\, z\ t.\ (z,t) \in \{(u,v)\ \mid\ & ((PC\ u \in_{sc} sc1) \wedge (u,\ sc1,\ v) \in \mathit{big\text{-}step} \vee \\ & (PC\ u \in_{sc} sc2) \wedge (u,\ sc2,\ v) \in \mathit{big\text{-}step})\}^* \wedge \\ & (sc1 \oplus sc2) \downarrow z\rangle\end{aligned}$$

We weaken the postcondition in a similar manner. Note that we need to add that the program counter does not point into the sequentially composed pieces of code anymore in order to apply the rule of consequence:

$$\begin{aligned}\langle\lambda\, z\ t.\ \neg\,(PC\ t \in_{sc} sc1) \wedge \neg\,(PC\ t \in_{sc} sc2) \wedge \\ (z,t) \in \{(u,v)\ \mid\ & ((PC\ u \in_{sc} sc1) \wedge (u,\ sc1,\ v) \in \mathit{big\text{-}step} \vee \\ & (PC\ u \in_{sc} sc2) \wedge (u,\ sc2,\ v) \in \mathit{big\text{-}step})\}^* \wedge \\ (sc1 \oplus sc2) \downarrow z\rangle.\end{aligned}$$

The destination triple now has the format required by the appropriate rule for sequential composition defined in our proof system. To apply this rule, we first have to provide an appropriate well-founded relation. We provide the following relation, which claims that the terminating executions of the sequential composition (which might 'circulate' through both parts of the composition) form a well-founded relation:

$$\begin{aligned}wf\{(t,s)\ \mid\ & (sc1 \oplus sc2) \downarrow s \wedge PC\ s \in_{sc} (sc1 \oplus sc2) \wedge \\ & (\ (PC\ s \in_{sc} sc1) \longrightarrow (s,\ sc1,\ t) \in \mathit{big\text{-}step} \wedge \\ & \ \ (PC\ s \in_{sc} sc2) \longrightarrow (s,\ sc2,\ t) \in \mathit{big\text{-}step}\ )\}\end{aligned}$$

To show that this relation is indeed well-founded, we use the following lemma which establishes that if a sequential composition $sc1 \oplus sc2$ is executed

starting in a state $f(k)$[1] and terminates, then for every possible execution path there exists an argument i such that the program counter in the state $f(i)$ does not point into the sequential composition anymore and therefore the execution terminates. Clearly, this reflects the characterization of a well-founded relation, i.e., that there is no infinitely going down chain. The resulting lemma is the following:

$$\begin{array}{l}\textbf{If } (sc1 \oplus sc2) \downarrow f\,k \textbf{ and} \\ \quad (\forall i.\ PC\ (f\,i) \in_{sc} (sc1 \oplus sc2) \Longrightarrow \\ \qquad (PC\ (f\,i) \in_{sc} sc1) \longrightarrow (f\,i,\ sc1,\ f\ (Suc\ i)) \in \mathit{big\text{-}step} \wedge \\ \qquad (PC\ (f\,i) \in_{sc} sc2) \longrightarrow (f\,i,\ sc2,\ f\ (Suc\ i)) \in \mathit{big\text{-}step}) \\ \quad \textbf{then } \exists i.\ \neg\ (PC\ (last\ (f\,i)) \in_{sc} sc1) \wedge \neg\ (PC\ (last\ (f\,i)) \in_{sc} sc2)\end{array}$$

We prove this lemma by induction on the termination predicate. Well-foundedness of the relation described above is then easily established. Now that the appropriate well-founded relation is specified, we have to show that the preconditions of the triples (1) and (2) from the assumptions of our overall proof goal can be strengthened to yield the specified invariant. Moreover, we need to show that the postconditions can be weakened to yield the invariant and that furthermore the state is decreased with respect to the provided well-founded relation. For (1) we need to show that we can obtain the following specification using the rule of consequence:

$$\begin{array}{l}\forall s'.\langle \lambda\, aux\ s.PC\ s \in_{sc} sc1 \wedge \\ \qquad\qquad (aux, s) \in \{(u,v).(PC\ u \in_{sc} sc1) \wedge (u,\ sc1,\ v) \in \mathit{big\text{-}step} \vee \\ \qquad\qquad\qquad\qquad (PC\ u \in_{sc} sc2) \wedge (u,\ sc2,\ v) \in \mathit{big\text{-}step}\}^* \wedge \\ \qquad (sc1 \oplus sc2) \downarrow aux \wedge s = s\langle\rangle \\ sc1 \\ \langle \lambda\, aux\ s.(aux, s) \in \{(u,v).(PC\ u \in_{sc} sc1) \wedge (u,\ sc1,\ v) \in \mathit{big\text{-}step} \vee \\ \qquad\qquad\qquad\qquad (PC\ u \in_{sc} sc2) \wedge (u,\ sc2,\ v) \in \mathit{big\text{-}step}\}^* \wedge \\ \qquad\qquad (sc1 \oplus sc2) \downarrow aux \wedge \\ \qquad\qquad ((sc1 \oplus sc2) \downarrow s' \wedge ((PC\ s' \in_{sc} sc1) \vee (PC\ s' \in_{sc} sc2)) \wedge \\ \qquad\qquad (PC\ s' \in_{sc} sc1) \longrightarrow (s',\ sc1,\ s) \in \mathit{big\text{-}step} \wedge \\ \qquad\qquad (PC\ s' \in_{sc} sc2) \longrightarrow (s',\ sc2,\ s) \in \mathit{big\text{-}step}))\}\end{array}$$

Note that this precondition is a conjunction consisting of a predicate referring to the program pointer (as required by the respective rule of the proof system) and the invariant. It also connects the states of the precondition to the states of the postcondition via the universally quantified variable s'. The postcondition in turn is a conjunction of the invariant and the requirement that the states of the postcondition are related to the states of the precondition (via s') with respect to the well-founded relation that we have given above. To prove that the specification can be derived from (1) we use the following auxiliary lemma:

[1] f is a function from the natural numbers into the states and not to be confused with the function f used in the definition of *do* f.

Lemma 3 (Transitivity of termination)
If $wff_{sc}\ (sc1 \oplus sc2)$ ***and*** $\cup_{sc}\ (sc1 \oplus sc2) \subseteq lis$ ***and*** $(sc1 \oplus sc2) \downarrow s$ ***and*** $s \neq s'$ ***and*** $PC\ s \in_{sc} (sc1 \oplus sc2)$ ***and***
$(s, s') \in \{(u, v) \mid (PC\ u \in_{sc} sc1) \wedge (u, sc1, v) \in big\text{-}step \vee (PC\ u \in_{sc} sc2) \wedge (u, sc2, v) \in big\text{-}step\}^*$ ***then*** $(sc1 \oplus sc2) \downarrow s'$

The lemma states that if execution of a sequential composition $sc1 \oplus sc2$ terminates starting in a state s, and furthermore s and s' are related via the relation that describes the intermediate states of executions which alternate between *sc1* and *sc2*, then execution of $sc1 \oplus sc2$ terminates from s' as well. Finally, the application of the rule for sequential composition can be applied to establish that the desired specification (3) is indeed valid in the calculus.

Once it is shown that the most general triple can be derived in the calculus, completeness follows directly (given that the code is well-formed and fulfills the requirements for labeled instruction sets):

Theorem 4 (Completeness for total correctness)
If $\cup_{sc}\ code \subseteq lis$ ***and*** $wff_{sc}\ code$ ***then***
$\models_t \langle P \rangle\ code\ \langle Q \rangle \longrightarrow \vdash_t \langle P \rangle\ code\ \langle Q \rangle$

Besides soundness and completeness of a proof system, completeness with respect to the underlying logical system (called relative completeness) and expressiveness are two important characteristics of a proof system. Because our proof calculus and the operational semantics are both specified in HOL, the completeness theorem ensures that any property provable on the basis of the operational semantics can also be established using our proof calculus. Expressiveness describes the ability of a logical system to express the intermediate predicates that may be necessary within a proof in our proof calculus. As pointed out in [Nip02], HOL can be used to formalize the completeness proof for our proof calculus and is therefore expressive with respect to our proof calculus. The weakest logical system that is expressive for our partial correctness calculus is first-order arithmetic because the most general triple can be expressed in it. For the total correctness calculus first-order arithmetic with least fixpoints is expressive. For a detailed discussion we refer to [Nip02].

In this section, we have shown that our extended calculus is sound and complete with respect to total correctness in the presence of unbounded nondeterminism. In the next section, we show how the calculus can be used for concrete verification efforts and how the mechanisms of Isabelle/HOL for the construction of well-founded relations can be exploited for this purpose.

5.5 Examples

In this section, we explain how our abstract formalization can be used for concrete total correctness proofs about low-level code. We show that the mechanisms provided by Isabelle/HOL for the construction of well-founded

relations are of great help. Furthermore, even the small examples we use here demonstrate that for reasoning about low-level code, machine assistance is important. The complexity already inherent to partial correctness proofs about unstructured code is elevated in the presence of variants that are needed for total correctness proofs. However, by having machine assistance available we can be sure that tricky corner cases are not overlooked in the proofs.

We instantiate the abstract type of state with a simple notion of concrete state given as a mapping *regs* of type $nat \Rightarrow nat$. One might think of such a function as assigning values from the natural numbers to registers identified by natural numbers.

To give a concrete meaning to our abstract state transformation instruction *do f*, we define HOL functions that operate on the previously defined notion of state. The first one assigns the value 5 to register number 1 and the second function increases the value in register 1 by one and sets the value in register two to four:

$ass5t1\ \ regs = \{regs(1 := 5)\}$
$incr14t2\ \ regs = \{regs(1 := (regs\ 1) + 1,\ 2 := 4\ )\}$

We first show how total correctness can be established for a sequential composition using a small example of straight line code. By straight line, we mean that the labels of the instructions are adjacent, i.e., there are no backwards jumps in the given code. We define the instruction set for the first example consisting of two applications of the *do* instruction using the state updates just defined as $instructionSetA = \{(1, (do\ ass5t1)), (2, do\ incr14t2)\}$.

As already explained above, to specify the behavior of a union of pieces of code, a function needs to be specified that assures termination of the sub pieces of code. For straight-line code such a variant can be easily given by defining a function that takes as input the program counter of a given state and returns its inverse with respect to the upper bound of the labels involved in the union operation. For our small example this function is defined as follows:

$lPC\ s = (if\ (PC\ s \geqslant 1 \wedge PC\ s \leqslant 3)\ then\ ((4::nat) - PC\ s)\ else\ 0)$

Building on this function we can define a well-founded relation. As noted in the Isabelle Tutorial [NPW02], proving that a relation is indeed a well-founded relation is a complicated task in general. However, it is often the case that potential relations are well-founded by construction. Isabelle provides support in terms of theorems for such constructions. The function **measure** (of type $('b \Rightarrow nat) \Rightarrow ('b \times 'b)\ set$) supports the construction of a well-founded relation given a function from an arbitrary type into the natural numbers. We define our well-founded relation using this command as: $pC = measure\ lPC$. Once the relation is defined, it is trivial to establish that pC is a well-founded relation. All that needs to be done is to unfold the definition of the relation. Theorems to establish well-foundedness are automatically provided to the simplifier when using the measure command.

We want to prove the following total correctness specification:

$$\vdash_t \langle \lambda aux\ s.\ PC\ s = 1 \rangle$$
$$(1 :: do\ ass5t1) \oplus (2 :: do\ incr14t2)$$
$$\langle \lambda aux\ s.\ PC\ s = 3 \wedge R\ s\ 1 = 6 \wedge R\ s\ 2 = 4 \rangle$$

To apply the rule for sequential composition, we first have to apply the rule of consequence. We use the following invariant *inv*:

$$\begin{aligned}\langle \lambda\, aux\ s.&PC\ s = 1 \vee \\ &PC\ s = 2\ \wedge R\ s\ 1 = 5\ \vee \\ &PC\ s = 3\ \wedge R\ s\ 1 = 6\ \wedge\ R\ s\ 2 = 4 \rangle\end{aligned}$$

Clearly, the precondition of the desired specification implies the invariant *inv* and the postcondition of the specification is implied by

$$\neg\ (PC\ s \in_{sc} (1 :: do\ ass5t1)) \wedge \neg\ (PC\ s \in_{sc} (2 :: do\ incr14t2)) \wedge inv.$$

We can now use the relation pC to prove the sequential composition. After application of the rule *HSEQ* (using the specified relation pC) from Figure 5.4 on page 79, we need to establish that both of the instructions involved in the sequential composition decrease the state with respect to the variant. This is also achieved by first using the rule of consequence and then unfolding the variants definition.

As a second example, we verify the total correctness of a program consisting of a loop. The low-level code we consider can be obtained by compiling the following high-level program, for example:

```
reg1 = {1,2,3,4};
while (reg1 < 10)
   {reg1 := reg1 + 1}.
```

When proving total correctness of such a (structured high-level) program one specifies a variant and needs to show that the body of the while loop strictly decreases the variant. In this case one takes the value of the variable *reg1* and takes the inverse of its value with respect to value mentioned in the loop condition. The low-level representation of the program can be given as:

```
1 do init1
2 brt sm10 3 5
3 do incr1
4 br 2
```

We use the following definitions to specify the initial non-deterministic assignment of a value from the set $\{1, 2, 3, 4\}$ to register 1 and the addition of 1 to the value residing in register 1 in line 3. Furthermore, the function *sm*10 defines the branch condition in line 2.

$init1\ regs = \{(regs(1:=1)),(regs(1:=2)),(regs(1:=3)),(regs(1:=4))\}$
$incr1\ \ regs = \{regs(1 := (regs\ 1) + 1)\}$
$sm10\ regs = (regs\ 1 < 10)$

Based on these definitions, we can now define the set of labeled instructions for the example. We then use this set to instantiate the abstract locale context (see Section 5.2) for a concrete setting. The assumptions of the locale context trivially hold.

We want to derive the following total correctness specification which states that executed from a state where the program pointer PC points to label 1, execution ends in a state where the PC points to 5 and register 1 holds the value 10:

$\langle \lambda aux\ s.\ PC\ s = 1 \rangle$
$\quad (1 :: do\ init1) \oplus ((2 :: brt\ sm10\ 3\ 5) \oplus ((3 :: do\ incr1) \oplus (4 :: br\ 2)))$
$\langle \lambda aux\ s.\ PC\ s = 5 \wedge R\ s\ 1 = 10 \rangle$

The proof tree is given in Figure 5.6 on page 93 . We start by explaining the variants we use in the proof and then explain the assertions used in the total correctness proof. To specify a variant for the piece of code we cannot solely use the value of the program counter. Due to the presence of the backward branching instruction at line 4, it is not possible to provide a suitable variant that only depends on the program counter. The variant needs to consider the value of the counter variable in register 1 ($R\ s\ 1$), which is incremented within the loop body. We now construct a suitable variant for the entire code using the lexicographical product of three variants that describe the behavior of the sequential compositions that the code is built from. We first define the variants in isolation. The first variant *v1rel* is for the sequential composition of the first instruction with the rest of the code. We specify a mapping *v1def* from the state to the natural numbers. Then, we use the Isabelle/HOL function measure, which based on this mapping constructs the well-founded relation *v1rel*.

$v1def\ s = (if\ (PC\ s = 1)\ then\ 1\ else\ 0)$
$v1rel = measure\ v1def$

The following two variants account for the behavior of the code that results from compilation of the while loop. The variant *v2rel* is defined as the inverse of the counter variable in register one with respect to its final value 10. The relation *v2rel* reflects the variant of the loop body from the high-level code. The variant *v3rel* is defined as the inverse of the program counter in a given state (with respect to the number of lines of code in our example).

$v2def\ s = (10 - (R\ s\ 1))$
$v2rel = measure\ v2def$
$v3def\ s = (if\ (PC\ s \geqslant 1 \wedge PC\ s \leqslant 5)\ then\ (6 - PC\ s)\ else\ 0)$
$v3rel = measure\ v3def$

The following variant *v4rel* is used only for the sequential composition of the last two instructions, which involve the instruction *br* 2 that sets the program counter to 2.

$$v4def\ s = (if\ (PC\ s = 3 \vee PC\ s = 4)\ then\ (5 - PC\ s)\ else\ 0)$$
$$v4rel = measure\ v4def$$

We can now define the desired well-founded relation as the inverse image of the function *cvdef*. This function maps a state to a tuple consisting of the three previously defined measure functions. The well-founded relation *cvrel* is the inverse image of the lexicographic product of the previously defined relations. Isabelle/HOL provides a predefined operator for this. The thus obtained relation is then well-founded by inheriting the well-foundedness results of the relations it is defined from.

$$cvdef\ s = ((v1def\ s),\ (v2def\ s)\ ,\ (v3def\ s))$$
$$cvrel = (inv\text{-}image\ (less\text{-}than\ <*lex*>\ less\text{-}than\ <*lex*>\ less\text{-}than)\ cvdef)$$

Now that we have defined the variants for the total correctness proof, we explain the assertions and invariants we use. The proof proceeds backwards from the total correctness specification mentioned above. The general proof strategy is to apply the rule of consequence in order to modify the pre- and postconditions of a given sequential composition so that a suitable invariant is obtained, which in turn enables the application of the rule for sequential composition.

We make the following abbreviations for structured code:

$$s234 = ((2 :: brt\ sm10\ 3\ 5) \oplus ((3 :: do\ incr1) \oplus (4 :: br\ 2)))$$
$$s34 = ((3 :: do\ incr1) \oplus (4 :: br\ 2))$$
$$s2 = (2 :: brt\ sm10\ 3\ 5)$$
$$s3 = (3 :: do\ incr1)$$
$$s4 = (4 :: br\ 2)$$

In the proof depicted in Figure 5.6 the following invariants are used. The invariants consist of assertions mentioning the program counter and the values in register 1 in the respective states. The latter reflects an invariant as it would be used in a high-level proof as well. The proof proceeds backwards from the desired total correctness specification using the respective invariants to derive specifications, which allow for the application of the rule of consequence.

$$\begin{aligned} Inv125\ aux\ s = (\lambda\, aux\ s.((PC\ s = 1) \vee \\ (PC\ s = 2 \wedge ((R\ s)\ 1) \geq 0 \wedge ((R\ s)\ 1) \leq 10) \vee \\ (PC\ s = 5 \wedge ((R\ s)\ 1) = 10)))\ aux\ s \end{aligned}$$

$$\begin{aligned} Inv235\ aux\ s = (\lambda\, aux\ s.((PC\ s = 2 \wedge ((R\ s)\ 1) \geq 0 \wedge ((R\ s)\ 1) \leq 10) \vee \\ (PC\ s = 3 \wedge ((R\ s)\ 1) < 10) \vee \\ (PC\ s = 5 \wedge ((R\ s)\ 1) = 10)))\ aux\ s \end{aligned}$$

The last invariant *Inv234* mentions the second part of the variant directly through the variable $s'a$. This is due to the backward jump as mentioned in the discussion of the variants.

$$\begin{array}{l} Inv234\ aux\ s\ s'a = \\ \quad (\lambda\, aux\ s.((PC\ s = 3 \wedge ((R\ s)1) < 10 \wedge ((R\ s)1) \leq 10) \wedge v2def\ s = v2def\ s'a \vee \\ \qquad\qquad (PC\ s = 4 \wedge ((R\ s)1) \leq 10) \wedge v2def\ s < v2def\ s'a \vee \\ \qquad\qquad (PC\ s = 2 \wedge ((R\ s)1) \leq 10) \wedge v2def\ s < v2def\ s'a))\ aux\ s \end{array}$$

To demonstrate how a proof is carried out using our mechanization, we now describe some of the proof steps used within the proof summarized in the proof tree in Figure 5.6. To apply the rule for sequential composition, we first apply the rule of consequence. We will use the invariant *Inv125* for the sequential composition of *1 :: do init1* and the rest of the code.

Note that we are doing a backward proof. So in order to apply the rule for sequential composition, the precondition $\langle \lambda aux\ s.\ PC\ s = 1 \rangle$ needs to imply the invariant and the postcondition needs to be implied by the invariant in conjunction with $\neg\ (PC\ s \in_{sc} (1 :: do\ init1)) \wedge \neg\ (PC\ s \in_{sc} ((2 :: brt\ sm10\ 3\ 5) \oplus ((3 :: do\ incr1) \oplus (4 :: br\ 2))))$. This is automatically proved by simplification. Now we can apply the rule for sequential composition by specifying the invariant and the combined variant *cvrel*. The proof obligation that *cvrel* is well-formed is again discharged by simplification. We obtain two subgoals. The first is the specification for the first instruction, where the PC in the precondition points to label 1:

$$\begin{array}{l} \forall\, s'.\langle \lambda aux\ s.\ (PC\ s \in_{sc} (1 :: do\ init1)) \wedge Inv125\ aux\ s \wedge s = s' \rangle \\ \qquad\qquad 1 :: do\ init1 \\ \quad \langle \lambda\, aux\ s.\ Inv125\ aux\ s\ \wedge (s, s') \in cvrel \rangle \end{array}$$

The second is the specification of the rest of the code where the PC points into the rest of the code.

$$\begin{array}{l} \forall\, s'.\langle \lambda aux\ s.\ PC\ s \in_{sc} ((2 :: brt\ sm10\ 3\ 5) \oplus ((3 :: do\ incr1) \oplus (4 :: br\ 2))) \wedge \\ \qquad\qquad Inv125\ aux\ s \wedge s = s' \rangle \\ \qquad (2 :: brt\ sm10\ 3\ 5) \oplus ((3 :: do\ incr1) \oplus (4 :: br\ 2)) \\ \quad \langle \lambda aux\ s.\ Inv125\ aux\ s \wedge (s,\ s') \in cvrel \rangle \end{array}$$

To show validity of the first specification, we use the rule of consequence and establish that $\langle \lambda aux\ s.\ PC\ s = 1 \rangle$ is implied by the precondition of the first specification and that

$$\langle \lambda aux\ s.\ PC\ s = 2 \wedge 0 \leqslant R\ s\ 1 \wedge R\ s\ 1 \leqslant 10 \rangle$$

implies the postcondition (which also mentions the relation of the precondition states and postcondition states with respect to the first part of the combined variant through the globally quantified variable s'). Again, the respective proof obligations are discharged by simplification. Afterwards, the specification is shown to be valid using the instantiation of the rule for *do f*.

The proof continues in this way, i.e., for the second specification we use an invariant that specifies the states with program counter value 2,3 and 5 and apply the rule for sequential composition. Note that we defined the variant so that the first component applies to the first instruction. The pre- and post-conditions were already strong enough to establish that pre- and poststates are related as required by the specified variant.

The remaining components of the combined variant can be regarded as being characteristic for the low-level representation of loops as explained in the following: The second component (*v2rel*) mentions the 'high-level' variant that depends on register 1 and the third component (*v3rel*) mentions the program counter with respect to all five lines of code.

In our small example, the loop body only consists of one instruction but the same technique also applies for loops that consist of more instructions in the body. Consider the two branching instructions *brt* (in line 2) and *br* (in line 4). Before and after the loop body (and also possible instructions within the loop body that do not alter the loop variant), the first and second components of the variant are equal for the states before and after these instructions, so the third component of the lexicographical product is considered. But considering the *br* instruction in line 4 this would not work, because it would not decrease the state in terms of the third component of the variant *v3rel*. This is exactly the problem when we arrive at the last sequential composition:

$$\begin{array}{l}\forall\, s'a.\\ \langle \lambda aux\ s.\ (PC\ s \in_{sc} ((3 :: do\ incr1) \oplus (4 :: br\ 2))) \wedge Inv235\ aux\ s \wedge s = s'a\rangle\\ \qquad\qquad (3 :: do\ incr1) \oplus (4 :: br\ 2)\\ \langle \lambda\, aux\ s.\ Inv235\ aux\ s\ \wedge (s, s'a) \in cvrel\rangle \end{array} \tag{i1}$$

Here, we cannot apply the rule of consequence using the invariant *Inv I234* defined as

$$\begin{array}{l}\langle \lambda\, aux\ s.((PC\ s = 3 \wedge ((R\ s)1) < 10) \vee\\ \qquad\qquad (PC\ s = 4 \wedge ((R\ s)1) \leq 10) \vee\\ \qquad\qquad (PC\ s = 2 \wedge ((R\ s)1) \leq 10)))\rangle \end{array} \tag{i2}$$

for the precondition and for the postcondition since these assertion are not strong enough to discharge the proof obligations resulting from an application of the rule of consequence. The reason is that in i2 only the assertion about the conjunct with the program counter pointing to 2 can hold since the other parts of *Inv234* are ruled out by the first conjunct about the program counter value not pointing to label 3 or 4. So considering i1 and i2 with respect to the combined variant *cvrel*, it cannot be established that the state is decreased for any component of the variant. The solution is to use the following invariant i3:

$$\begin{array}{l}\langle \lambda\, aux\ s.((PC\ s = 3 \wedge ((R\ s)1) \leq 10) \wedge v2defs = v2defs'a \vee\\ \qquad\qquad (PC\ s = 4 \wedge ((R\ s)1) \leq 10) \wedge v2defs < v2defs'a \vee\\ \qquad\qquad (PC\ s = 2 \wedge ((R\ s)1) \leq 10) \wedge v2defs < v2defs'a))\rangle \end{array} \tag{i3}$$

In this section, we have demonstrated the concrete application of our mechanized proof calculus using a small example. We have shown that concrete proofs benefit from our mechanization because the mechanisms of Isabelle/HOL for constructing well-founded relations can be exploited. Furthermore, even these minimalistic examples show that for proofs about low-level code, machine assistance is of great help in order to cope with the complexity of such proofs.

5.6 Summary

In this chapter, we have formalized a basic low-level language in Isabelle/HOL. Although the language is minimalistic, it is highly expressive and capable of expressing a variety of concepts including instructions that might introduce non-determinism and unrestricted jumps between labeled instructions. Based on the ideas from [SU05], we have formalized a compositional big-step semantics and related Hoare logics for the language. To reason about functional properties and termination behavior, we have defined a total correctness logic and established soundness and completeness for the language. In the next chapter, we enrich our basic low-level language with real-time capabilities. Furthermore, we extend the language with instructions that allow for communication with the environment. Using these extensions, we then develop a strategy for the specification and verification of conformance relations between high-level specifications and low-level implementations in a real-time setting.

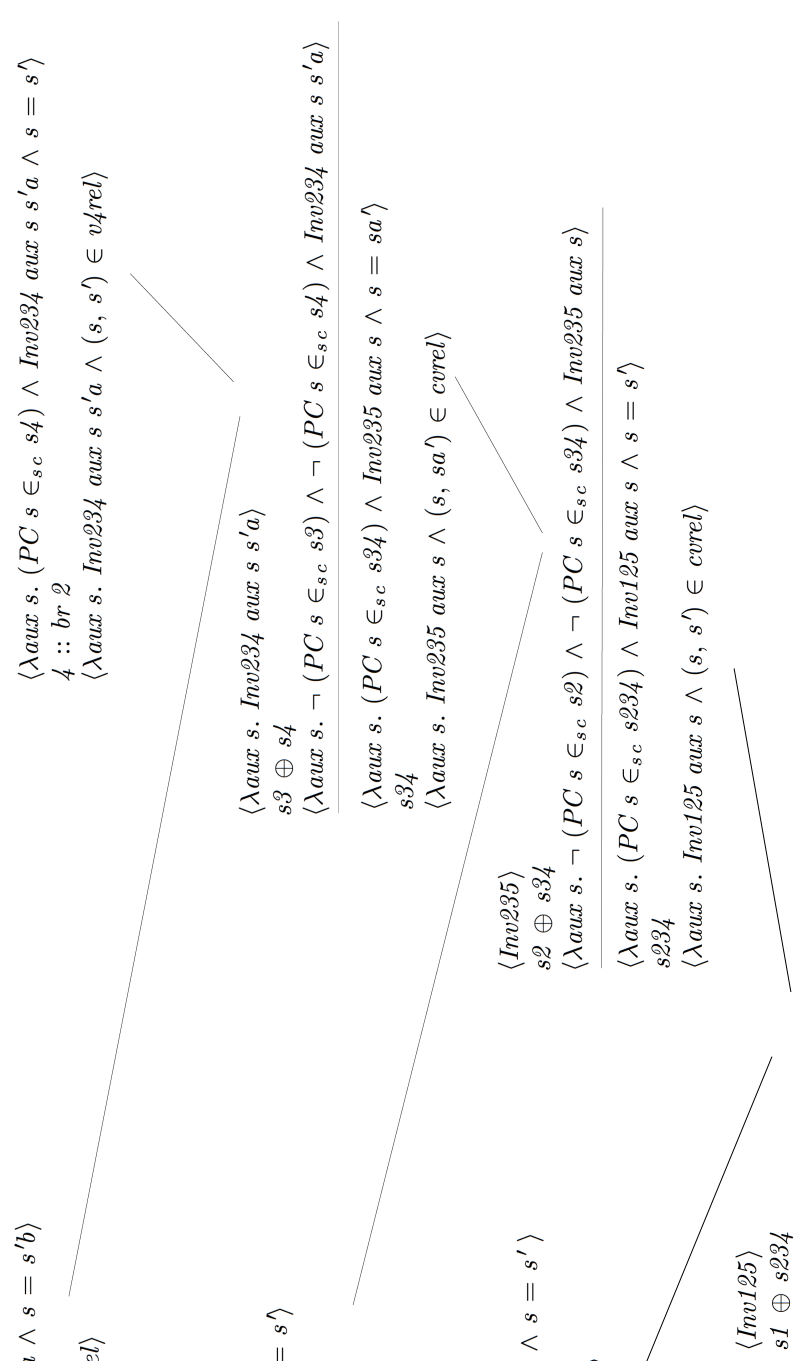

Figure 5.6: Proof Tree for the Loop Example

6 A Semantic Stack for Real-Time Low-Level Code

In the last chapter, we have presented mechanized small-step and big-step semantics for our basic low-level language and a dedicated proof logic. It enables the verification of functional correctness properties about sequential low-level code. Additionally, the total correctness logic makes proofs about the termination behavior of low-level code possible. The work presented in the last chapter focused on the internal behavior of low-level code, i.e., changes to the internal state. Building on this, in this chapter we integrate externally visible (or process-specific) behavior into our verification environment. This reflects information about communications with the environment as well as timing intervals between such externally visible communications. Our treatment of communications resembles the way communication and synchronization is modeled in CSP-like languages.

Our main contribution presented in this chapter is the extension of our untimed low-level language from the last chapter to a timed language. For the instructions presented in the last chapter this implies that execution now takes a certain amount of time. Furthermore, we add an instruction that lets time pass by and instructions that enable the low-level code to synchronize with its environment, i.e., to receive values or to output values through channels. We start by defining a small-step semantics for the low-level language. During the execution of low-level code, it is now possible to observe that either communications take place, or that time may pass by. To reflect this behavior in our semantics, transitions are either labeled with events that reflect internal behavior, externally visible communications or that model the passage of time. Consequently, the obtained small-step semantics can be interpreted in terms of a timed labeled transition system. Based on the small-step semantics, we define a big-step semantics that makes it possible to model the semantics of executions in a compositional manner. As in the last chapter, we also define a proof logic for terminating executions. Here, the possible communication of events throughout the execution of code is captured in terms of a list of events that each have a timestamp referring to the point of time during execution at which they could be observed. In the next chapter, we use the semantics and proof calculus presented in this chapter to enable conformance proofs about low-level code with respect to process-algebraic specifications.

6.1 A Timed Communicating Low-Level Language

To extend our basic low-level language with a notion of time and communication capabilities, we first extend the type of state and the type of instructions to support the desired extensions. In the next sections, we then extend the semantics and the proof calculus.

6.1.1 Syntax and State Definitions

The notion of state is again polymorphic with respect to the actual store that programs operate on. The program counter is a label from the natural numbers. To keep track of the passage of time within our semantics, we extend the state with a clock variable *tnow*. To prepare for a later extension to a distributed setting (explained briefly in Section 6.5), we extend the state with a natural number *PId* that can be interpreted as a unique process ID. The selectors *PC*, *PId* and *Ptnow* can be used for the respective parts of the state.

$$\begin{aligned}\textbf{record}\ (\prime a)\ state = \ & R \ :: \prime a \ \ (\text{-}_R) \\ & PC \ :: label \ \ (\text{-}_{PC}) \\ & PId \ :: nat \ \ (\text{-}_{PId}) \\ & tnow :: real \ \ (\text{-}_{Ptnow})\end{aligned}$$

We introduce instructions for communication and for explicit handling of time. Note that in contrast to the last chapter, the instructions are not only polymorphic in the state *'a* but also in the type of events *'b*, which are used for communication purposes. Thus, for concrete scenarios the structure of communication events can be defined as needed.

$$\begin{aligned}\textbf{datatype}\ (\prime a, \prime b)instr = \ & out\ (\prime a \Rightarrow \prime b)\ (\prime a \Rightarrow real)\ label \mid \\ & input\ (\prime a \Rightarrow \prime b\ set)\ (\prime a \Rightarrow \prime b \Rightarrow \prime a)\ (\prime a \Rightarrow real)\ label \mid \\ & br\ label \mid \\ & brt\ (\prime a \Rightarrow bool)\ label\ label \mid \\ & do\ (\prime a \Rightarrow ((\prime a \times label)\ set)) \mid \\ & wait\ real \mid \\ & endInstruction\end{aligned}$$

For communication with the environment, the instruction *out* can communicate a visible event that might depend on the current state. The *input* instruction offers the environment a choice between different events in dependence of the current state and changes the internal state of a component dependent on the event it synchronized on with its environment. The state change is realized using a function that maps a state and a given event to a new state. Furthermore, for the instructions *out* and *input*, a timeout can be specified. If the environment is not ready to participate in the communication of the specified event or set of events, execution proceeds at a specified timeout label that resembles exception handling in case of communication timeouts. The branching instructions *br* and *brt* are defined as in the last chapter. The instruction *do f* is also defined similarly to the same instruction from the last

chapter but here it is allowed to change the program counter[1]. It formalizes that a function f is applied to the current state and might change the store and the program counter. Again, the function might yield a set of states. The *endInstruction* is used to signal to the environment that the execution of the component has successfully terminated (in contrast to evaluations that get stuck because the program pointer does not point to an instruction anymore). The *wait* instruction lets a specified amount of time pass by.

6.2 Timed Small-Step Semantics

In this section, we define the timed small step semantics for the low-level language. The main idea is that individual instructions correspond to Timed CSP processes. Consequently, we model the semantics of the low-level language like an external observer would witness the behavior of a component over time. Modeling the semantics in such a Timed CSP-oriented way is motivated by the goal of relating low-level code to process specifications. By defining an equivalent semantic basis for both languages, we are able to use this as the basis for conformance proofs (as presented in Chapter 7). In general, execution of an instruction in our setting involves the following steps:

- Time passes by when the instruction is prepared to be executed.
- If the instruction is a communication instruction, further time may pass by while the instruction is waiting for a communication with the environment. If a timeout is specified this waiting period is bounded.
- The effect of the instruction takes place. It may either be visible to the observer through the communication of an event (in case of a communication instruction) or not visible (in case of an internal instruction). The effect of instructions that realize internal behavior is observed from the outside as a τ event.
- After the effect of the instruction has been observed, time passes in order to store results of possible variable assignments.
- When the execution of an instruction is finished, the program pointer might point to a further instruction or, if not, execution stops and only time can pass by.

To keep our formalization in a reasonable size, we follow the common abstraction in CSP that the communication of visible events itself happens instantaneously, i.e., we do not explicitly model the duration of such events using, for example start and stop events. To model the different phases of executing an instruction and to assure that no unwanted transitions are possible between the phases of execution, we use the following type constructors:

[1] This allows a more compact specification of internal non-determinism in our timed low-level language.

datatype $('a)extstate = nstate\ \ ('a)state\ |$
$tbstate\ \ ('a)state \times real\ |$
$evdelstate\ \ ('a)state\ |$
$evstate\ \ \ ('a)state\ \ |$
$tostate\ \ ('a)state \times real\ |$
$tastate\ \ ('a)state \times real\ |$
$tsstate\ \ ('a)state$

The different types of states are used in the definition of the timed small-step semantics to ensure that no irregular transitions are possible. The definition of *('a)extstate* builds on the notion of state *('a)state*, which we have introduced in the beginning of this chapter in Section 6.1.1. The different types of states act as wrapper states around the basic state definition and guide the timing and communication behavior of a low-level program in the definition of the small-step semantics given below. The type of *nstate* corresponds to the basic state, *tbstate* and *tastate* are states from which time can pass by before and after the possibly visible effect of an instruction. Therefore, these states are defined not merely as a wrapper state around *('a)state* but also contain a countdown variable, which is used to keep track of the passage of time. The same intuition holds for *tostate*. This type of state is used to keep track of the passage of time if a timeout value is specified. The *evdelstate* is used if a communication instruction has to wait for the environment so that communication is delayed, while from *evstate* only the communication of events (including the internal event τ) is possible. If a program has terminated, the termination state *tsstate* is used to ensure that no further event transitions are possible, i.e., that only time can advance.

The small-step semantics is of the following type:

$(('a)extstate \times ('b)eventplus \times ('a)extstate)set)$.

Note that this can be interpreted as a labeled transition system. The labels are defined using the datatype $('b)eventplus$ (parameterized over the abstract type of events $'b$ used in the definition of the communication instructions). The datatype $('b)eventplus$ is defined as follows:

datatype $'b\ \ eventplus = time\ \ real\ \ |\ \ ev\ \ 'b\ \ |\ \ \checkmark\ \ |\ \ \tau$

It comprises the time steps (*time*), visible events (*ev*), the special event for successful termination ($\checkmark$) and the internal event (τ). The time related transition rules depicted in Figure 6.1 define the behavior of instructions, while the rules given in Figure 6.3 model the actual effect an instruction has on the current state. The time related rules depend on two functions called *tibef* and *tiaft* which are fixed in the locale context *llvm* that the rules are defined in. The first yields the time that elapses before the effect of an instruction while the latter yields the amount of time that passes by after the visible (or internal) effect of the respective instruction. The functions are of type $('a,'b)instr \Rightarrow nat \Rightarrow real$ and it is furthermore required that they yield positive values greater zero. This is necessary in order to interpret the semantics as a timed labeled

$$\frac{(s_{PC},\ ins) \in lis\ s_{PId} \qquad tibef\ ins\ s_{PId} = ta + tx \qquad 0 < ta \qquad 0 \leqslant tx \qquad s' = s(\!|tnow := s_{Ptnow} + ta|\!)}{(nstate\ s,\ time\ ta,\ tbstate\ (s',\ tx)) \in tsmallstep}\text{NSTEP}$$

$$\frac{0 < ta \qquad 0 \leqslant tx' \qquad tx = ta + tx' \qquad s' = s(\!|tnow := s_{Ptnow} + ta|\!)}{(tbstate\ (s,\ tx),\ time\ ta,\ tbstate\ (s',\ tx')) \in tsmallstep}\text{TBSTEP}$$

$$\frac{\begin{array}{r}(s_{PC},\ do\ f) \in lis\ s_{PId} \vee (s_{PC},\ wait\ ti) \in lis\ s_{PId} \vee (s_{PC},\ br\ m) \in lis\ s_{PId} \\ \vee (s_{PC},\ brt\ b\ m\ n) \in lis\ s_{PId} \vee (s_{PC},\ endInstruction) \in lis\ s_{PId}\end{array}}{(tbstate\ (s,\ 0),\ \tau,\ evstate\ s) \in tsmallstep}\text{TBEVSTEP}$$

$$\frac{\begin{array}{r}((s_{PC},\ input\ c\ f\ fto\ tol) \in lis\ s_{PId} \vee (s_{PC},\ out\ f'\ fto\ tol) \in lis\ s_{PId}) \wedge fto\ s_R = 0 \\ \vee (s_{PC},\ endInstruction) \in lis\ s_{PId}\end{array}}{(tbstate\ (s,\ 0),\ \tau,\ evdelstate\ s) \in tsmallstep}\text{TBDLSTEP}$$

$$\frac{0 < ta \qquad s' = s(\!|tnow := s_{Ptnow} + ta|\!)}{(evdelstate\ s,\ time\ ta,\ evdelstate\ s') \in tsmallstep}\text{DLDLSTEP}$$

$$\frac{(evstate\ s,\ e,\ tastate\ (t,\ tx')) \in tsmallstep \qquad \forall\, ta.\ e \neq time\ ta}{(evdelstate\ s,\ e,\ tastate\ (t,\ tx')) \in tsmallstep}\text{EVDELTASTEP}$$

$$\frac{\begin{array}{r}(s_{PC},\ input\ c\ f\ fto\ tol) \in lis\ s_{PId} \vee (s_{PC},\ out\ f'\ fto\ tol) \in lis\ s_{PId}) \\ \wedge\ 0 < fto\ s_R \wedge fto\ s_R = tx\end{array}}{(tbstate\ (s,\ 0),\ \tau,\ tostate\ (s,\ tx)) \in tsmallstep}\text{TBTOSTEP}$$

$$\frac{0 < ta \qquad 0 \leqslant tx' \qquad tx = ta + tx' \qquad s' = s(\!|tnow := s_{Ptnow} + ta|\!)}{(tostate\ (s,\ tx),\ time\ ta,\ tostate\ (s',\ tx')) \in tsmallstep}\text{TOTOSTEP}$$

$$\frac{(evstate\ s,\ e,\ tastate\ (t,\ tx')) \in tsmallstep \qquad \forall\, ta.\ e \neq time\ ta}{(tostate\ (s,\ tx),\ e,\ tastate\ (t,\ tx')) \in tsmallstep}\text{TOTAESTEP}$$

$$\frac{\begin{array}{r}((s_{PC},\ input\ c\ f\ fto\ tol) \in lis\ s_{PId} \vee (s_{PC},\ out\ f'\ fto\ tol) \in lis\ s_{PId}) \wedge tl = l) \\ t = s(\!|PC := l|\!) \qquad l \neq 0 \qquad l \neq s_{PC} \\ \exists\, ins.\ (s_{PC},\ ins) \in lis\ s_{PId} \wedge tx = tiaft\ ins\ s_{PId}\end{array}}{(tostate\ (s,\ 0),\ \tau,\ tastate\ (t,\ tx)) \in tsmallsteptotatstep}\text{TOTASTEP}$$

$$\frac{0 < ta \qquad 0 \leqslant tx' \qquad tx = ta + tx' \qquad t' = t(\!|tnow := t_{Ptnow} + ta|\!)}{(tastate\ (t,\ tx),\ time\ ta,\ tastate\ (t',\ tx')) \in tsmallstep}\text{TATASTEP}$$

$$(tastate\ (t,\ 0),\ \tau,\ nstate\ t) \in tsmallstep\text{TATNSTEP}$$

$$\frac{(\forall\, ins.\ (t_{PC},\ ins) \notin lis\ t_{PId}) \vee t_{PC} = 0}{(nstate\ t,\ \tau,\ tsstate\ t) \in tsmallstep}\text{NSTSTOP}$$

$$\frac{0 < ta \qquad t_{PC} = 0 \vee (\forall\, ins.\ (t_{PC},\ ins) \notin lis\ t_{PId}) \qquad t' = t(\!|tnow := t_{Ptnow} + ta|\!)}{(tsstate\ t,\ time\ ta,\ tsstate\ t') \in tsmallstep}\text{TSTSSTEP}$$

Figure 6.1: Time Related Transition Rules

transition system. The timing functions yield a timing value dependent on the instruction (and its arguments)[2] to be executed and the process ID.

[2]The behavior of the instruction *wait* is realized using the timing functions. Given the argument of the instruction the timing functions ensure that the specified amount of time passes by.

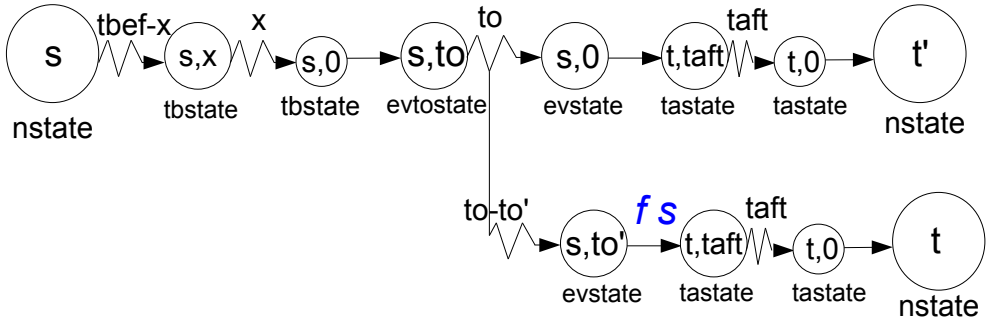

Figure 6.2: Labeled transtion system for the `out` instruction

We explain the intuition behind the rules using the instruction `out` with a specified timeout value as an example. In Figure 6.2, the LTS corresponding to the rules is shown. If the instruction pointer in a state s points to an `out` instruction from a 'normal' state (*nstate*), first the amount of time given by the function *tibef* passes by (rule *nstep*). Within the rules of the small-step semantics this is realized using a transition from *nstate* to *tbstate*, where the 'countdown' variable tx is set to the value given by *tibef*. The passage of time is then realized by transitions from *tbstate* to *tbstate* where the countdown variable is decreased by the amount of time ta that passes by (rule *tbstep*). These transitions are timed transitions (denoted by the jagged arrows). Once the countdown variable is decreased to zero, an internal transition to *evtostate* happens (rule *tbtostep*) since a timeout to greater than zero is specified and the current instruction is a communication instruction as required by the premise of the inference rule. The value of to is used as a countdown variable to track the time that elapses while the `out` instruction waits for communication with the environment (rule *totostep*). As long as the timeout has not elapsed, it is possible to communicate the desired event e by taking a transition directly to *tastate* (rule *totaestep*). As the premise of the inference rule indicates that the event e and the resulting state t is obtained by the state transition rule *sout*. The event is given by the function f dependent on the state s and t is obtained from s by increasing the program counter by one. If the timeout has elapsed and the countdown variable has been decreased to zero, an internal transition is taken and the program counter is set to the specified timeout label to (rule *totastep*). In both cases, the respective transition leads to a *tastate*, which is used to let time pass by after the `out` instruction. Similar to the *tbstate* a countdown variable is used and the transition is a timed transition. Once the countdown variable is decreased to zero, an internal transition to the next normal state t is possible. From a normal state, another instruction can be executed if the program points to one. If not, an internal transition to the termination state *tsstate* happens. In the termination state only time can pass by. The premises of the inference rules *nststop* and *tstsstep* ensure that the program counter in state t does not point to an instruction or that it points to the special label 0[3].

[3]We use the label 0 to model successful termination, i.e., when a program terminates successfully by executing the instruction *endInstruction*, the program counter is set to 0.

$$\frac{(s_{PC},\ do\ f) \in lis\ s_{PId} \qquad 0 < m \qquad (x,\ m) \in f\ s_R \qquad m \neq s_{PC} \qquad tiaft\ (do\ f)\ s_{PId} = tx \qquad t = s(\!|R := x,\ PC := m|\!)}{(evstate\ s,\ \tau,\ tastate\ (t,\ tx)) \in tsmallstep}\text{SDOF}$$

$$\frac{0 < ti \qquad (s_{PC},\ wait\ ti) \in lis\ s_{PId} \qquad tiaft\ (wait\ ti)\ s_{PId} = tx \qquad t = s(\!|PC := s_{PC} + 1|\!)}{(evstate\ s,\ \tau,\ tastate\ (t,\ tx)) \in tsmallstep}\text{PWAIT}$$

$$\frac{(s_{PC},\ out\ f\ to\ tol) \in lis\ s_{PId} \qquad to\ s_R = 0 \wedge tol = 0 \vee 0 < to\ s_R \wedge 0 < tol \qquad f\ s_R = e \qquad tiaft\ (out\ f\ to\ tol)\ s_{PId} = tx \qquad t = s(\!|PC := s_{PC} + 1|\!)}{(evstate\ s,\ ev\ e,\ tastate\ (t,\ tx)) \in tsmallstep}\text{SOUT}$$

$$\frac{(s_{PC},\ input\ ef\ f\ to\ tol) \in lis\ s_{PId} \qquad e \in ef\ s_R \qquad to\ s_R = 0 \wedge tol = 0 \vee 0 < to\ s_R \wedge 0 < tol \qquad x = f\ s_R\ e \qquad tiaft\ (input\ ef\ f\ to\ tol)\ s_{PId} = tx \qquad t = s(\!|R := x,\ PC := s_{PC} + 1|\!)}{(evstate\ s,\ ev\ e,\ tastate\ (t,\ tx)) \in tsmallstep}\text{SINPUT}$$

$$\frac{(s_{PC},\ br\ m) \in lis\ s_{PId} \qquad s_{PC} \neq m \qquad 0 < m \qquad tiaft\ (br\ m)\ s_{PId} = tx \qquad t = s(\!|PC := m|\!)}{(evstate\ s,\ \tau,\ tastate\ (t,\ tx)) \in tsmallstep}\text{SMBR}$$

$$\frac{(s_{PC},\ brt\ b\ m\ n) \in lis\ s_{PId} \qquad b\ s_R \qquad s_{PC} \neq m \qquad 0 < m \qquad tiaft\ (brt\ b\ m\ n)\ s_{PId} = tx \qquad t = s(\!|PC := m|\!)}{(evstate\ s,\ \tau,\ tastate\ (t,\ tx)) \in tsmallstep}\text{SMBRTT}$$

$$\frac{(s_{PC},\ brt\ b\ m\ n) \in lis\ s_{PId} \qquad \neg\ b\ s_R \qquad s_{PC} \neq n \qquad 0 < n \qquad tiaft\ (brt\ b\ m\ n)\ s_{PId} = tx \qquad t = s(\!|PC := n|\!)}{(evstate\ s,\ \tau,\ tastate\ (t,\ tx)) \in tsmallstep}\text{SMBRTF}$$

$$\frac{(s_{PC},\ endInstruction) \in lis\ s_{PId} \qquad tiaft\ endInstruction\ s_{PId} = tx \qquad t = s(\!|PC := 0|\!)}{(evstate\ s,\ \surd,\ tastate\ (t,\ tx)) \in tsmallstep}\text{SENDINSTR}$$

Figure 6.3: State Related Transition Rules

In the given example, a timeout value was specified for the `out` instruction. If a value of zero is specified for a communication instruction, then the communication of the respective event can be delayed arbitrarily long by the environment. This is realized using the rule *tbdlstep*. From the *evdelstate* the communication of events can be delayed for any interval *ta*. At any point of time, a transition with a communication of the respective event is possible if it had been possible from an *evstate* (required by the premise of the rule *evdeltastep*). For instructions realizing internal behavior, an immediate internal transition to the *evstate* is possible after the time before the instruction has elapsed (rule *tbevstep*). From the *evstate* a transition to a *tastate* is possible, as allowed by the state transition related rules given in Figure 6.3.

The well-formedness predicate for sets of labeled instructions ensures that no instruction is labeled with 0.

6.2.1 Properties of the Sequential Semantics

To interpret the rules of the extended semantics in terms of a labeled transition system, we need to show that the semantics fulfills the well-formedness conditions given below.

The first lemma states that if time can advance in our semantics, the time interval is greater than zero:

Lemma 4 (Time positive for timed small-step)
If $(s, time\ d1, t) \in tsmallstep$ ***then*** $0 < d1$

The following property states that the passage of time is deterministic, i.e., that the passage of time does not introduce any non-determinism.

Lemma 5 (Time determinism for timed small-step)
If $(s, time\ d, t) \in tsmallstep$ ***and*** $(s, time\ d, u) \in tsmallstep$ ***then*** $t = u$

The next two lemmas establish important properties that allow us to interpret the semantics as a labeled transition system. First, we can regard any two consecutive time steps as one big time step.

Lemma 6 (Time additivity for timed small-step)
If $(s, time\ d1, s') \in tsmallstep$ ***and*** $(s', time\ d2, t) \in tsmallstep$ ***then*** $(s, time\ (d1 + d2), t) \in tsmallstep$

Second, for any time step of any duration there exists a state so that the time step can be split.

Lemma 7 (Time Interpolation for timed small-step)
If $(s, time\ (d1 + d2), t) \in tsmallstep$ ***and*** $0 < d1$ ***then*** $\exists\, s'.\ (s, time\ d1, s') \in tsmallstep \wedge (s', time\ d2, t) \in tsmallstep$

For the timed labeled small-step semantics presented in this section, we define a corresponding big-step semantics in the next section. The goal is to reduce the complexity when analyzing low-level code. Complexity is mainly introduced by the high amount of transition rules needed to model the timing behavior in the small-step semantics. In the next section, we therefore model the timing behavior of low-level code in a more compact way using a big-step semantics. Furthermore, we also show that information about program behavior can be transferred to the level of the small-step semantics.

6.3 Big-Step Semantics

In this section, we define a big-step semantics for our low-level real-time language. Following the strategy of the last chapter, we impose a structure on sets of instructions to obtain a compositional semantics.

6.3.1 State Definitions and Semantics

Since our low-level real-time language provides communication mechanisms that enable low-level code to communicate with its environment, the semantics needs to reflect this behavior. In our big-step semantics, communications are captured using timed traces. Our timed traces are lists that consist of tuples of type *real* × *('b eventplus)*. The first part of the tuple is a timestamp that captures at which point in time the event occurred. The second part of the tuple is the event itself. Note that here the point of time given by a timestamp is not relative to the beginning of the state at which execution of the respective piece of code starts. Rather the timestamps are total. The big-step semantics relates pairs of states and traces defined as follows:

type-synonym $('a,'b)stateTrace = ('a)state \times ('b)ttrace$

The intuition is that the execution of structured code starts in a state where the execution might already have accumulated a communication trace. Starting from the given state, communication events that occur while executing the specified section of code are appended to the already given timed trace. If execution terminates, the semantics yields the respective state and the generated trace. Thus, the big-step semantics is of type

$$(('a,'b)stateTrace \times ('a,'b)pstructuredCode \times ('a,'b)stateTrace)set.$$

The type of structured code is defined as in Chapter 5 but is now indexed by the process id of a process. In Figure 6.4, an excerpt of rules of the big-step semantics is given. The first rule is the one for the instruction *do f*. Since the command is not performing any communication, the communication trace is not altered in the conclusion of the inference rule. The state is updated for every pair consisting of a new store and new program pointer given by the function f applied to the store R in the state s. Furthermore, the value of the clock in state t is updated according to the timing functions *tibef* and *tiaft* as introduced in Section 6.2. Since *do f* now changes the program counter, the rule is only applicable if the target label is not equal to the program counter of s. Otherwise, the instruction would block. The rules *OUT* and *INPUT* are concerned with `input` and `output` instructions. Since the `input` and `output` instructions realize communication with the environment, a transition also alters the communication trace. In our big-step semantics, a single step corresponds to various steps performed on the level of the small-step semantics. This is reflected by the respective rules for the `input` and `output` instructions. The clock value is updated by adding the amount of time that passes before and after the visible effect of the instruction and the amount of time tx that the environment delayed the communication. In the communication trace though, the time stamp for the communicated event is obtained from the original clock value in state s by adding the waiting time before the instruction and the delay time that is due to the environment. A tuple consisting of event e and the appropriate timestamp is then appended to the communication trace *trs* of the origin state. The store, i.e., the values of program values, is updated similar to the small-step semantics. For the instruction *endInstruction*, it is

$$\frac{\begin{array}{c} wff_{sc}\ sc \qquad (l,\ do\ f) \in lis\ p1 \qquad sc = (l_{::} do\ f) \\ s_{PC} = l \qquad s_{PId} = p1 \qquad (x,\ m) \in f\ s_R \qquad m \neq l \qquad m \neq 0 \\ t = s(\!|R := x,\ PC := m,\ tnow := s_{Ptnow} + tibef\ (do\ f)\ p1 + tiaft\ (do\ f)\ p1|\!) \end{array}}{((s,\ trs),\ {}_p1_{|}sc,\ t,\ trs) \in big\text{-}step}\ \text{DOF}$$

$$\frac{\begin{array}{c} wff_{sc}\ sc \qquad (l,\ out\ f\ fto\ tol) \in lis\ p1 \qquad fto\ s_R = 0 \qquad tol = 0 \\ sc = (l_{::} out\ f\ fto\ tol) \qquad s_{PC} = l \qquad s_{PId} = p1 \qquad f\ s_R = e \\ 0 < tx \qquad tb = tibef\ (out\ f\ fto\ tol)\ p1 \qquad ta = tiaft\ (out\ f\ fto\ tol)\ p1 \\ t = s(\!|PC := s_{PC} + 1,\ tnow := s_{Ptnow} + tb + tx + ta|\!) \\ trt = trs \bullet [(s_{Ptnow} + (tb + tx),\ ev\ e)] \end{array}}{((s,\ trs),\ {}_p1_{|}sc,\ t,\ trt) \in big\text{-}step}\ \text{OUT}$$

$$\frac{\begin{array}{c} wff_{sc}\ sc \qquad (l,\ input\ ef\ f\ fto\ tol) \in lis\ p1 \\ 0 < tx \qquad fto\ s_R = 0 \qquad tol = 0 \qquad sc = (l_{::} input\ ef\ f\ fto\ tol) \\ e \in ef\ s_R \qquad x = f\ s_R\ e \qquad s_{PC} = l \qquad s_{PId} = p1 \\ tb = tibef\ (input\ ef\ f\ fto\ tol)\ p1 \qquad ta = tiaft\ (input\ ef\ f\ fto\ tol)\ p1 \\ t = s(\!|PC := s_{PC} + 1,\ R := x,\ tnow := s_{Ptnow} + tb + tx + ta|\!) \\ trt = trs \bullet [(s_{Ptnow} + (tb + tx),\ ev\ e)] \end{array}}{((s,\ trs),\ {}_p1_{|}sc,\ t,\ trt) \in big\text{-}step}\ \text{INPUT}$$

$$\frac{\begin{array}{c} wff_{sc}\ sc \qquad (l,\ endInstruction) \in lis\ p1 \\ sc = (l_{::} endInstruction) \qquad s_{PC} = l \qquad s_{PId} = p1 \\ 0 < tx \qquad tb = tibef\ endInstruction\ p1 \qquad ta = tiaft\ endInstruction\ p1 \\ t = s(\!|PC := 0,\ tnow := s_{Ptnow} + tb + tx + ta|\!) \\ trt = trs \bullet [(s_{Ptnow} + (tb + tx),\ \surd)] \end{array}}{((s,\ trs),\ {}_p1_{|}sc,\ t,\ trt) \in big\text{-}step}\ \text{END}$$

$$\frac{\begin{array}{c} sc = (sc1_{\oplus} sc2) \qquad s_{PC} \in_{sc} sc1 \qquad wff_{sc}\ sc \\ s_{PId} = p1 \qquad s1_{PId} = p1 \qquad ((s,\ trs),\ {}_p1_{|}sc1,\ s1,\ trs1) \in big\text{-}step \\ ((s1,\ trs1),\ {}_p1_{|}(sc1_{\oplus} sc2),\ t,\ trt) \in big\text{-}step \end{array}}{((s,\ trs),\ {}_p1_{|}sc,\ t,\ trt) \in big\text{-}step}\ \textsc{Seq1}$$

$$\frac{\neg\ (s_{PC} \in_{sc} sc) \qquad wff_{sc}\ sc \qquad s_{PId} = p1}{((s,\ trs),\ {}_p1_{|}sc,\ s,\ trs) \in big\text{-}step}\ \textsc{Term}$$

Figure 6.4: Excerpt from the Rules of the Big-Step Semantics

possible to delay the communication of successful termination for an arbitrary amount of time. The rule for sequential composition *SEQ*1 is defined similar to the rule explained in Section 5.4.2. However, here it is also required to provide an intermediate communication trace *trs*1 for the sequential composition. The rule for unsuccessful termination *TERM* is defined similar to the one in Section 5.4.2.

6.3.2 Relation to the Timed Small-step Semantics

To be able to transfer verification results from the level of our big-step semantics to the level of our small-step semantics, we show that the two semantics agree with respect to executions leading from a *nstate* to another *nstate*. We

concentrate on these executions since on the level of the big-step semantics there are no intermediate states when considering a single step from some state s to some other state t. Furthermore, in our big-step semantics no time passes by when the program pointer does not point to an instruction in the piece of structured code to be evaluated anymore, i.e., there is no *tsstate* on the level of the big-step semantics.

We use an intermediate relation *texec* that reflects the maximal execution of a number of small-steps with respect to a given set of instructions. For a given unstructured set of instructions, this relation describes the executions that end in a state where the program pointer does not point into the set of instructions anymore. A single step of the big-step semantics corresponds to a series of transitions in the small-step semantics where communications can take place during execution. The relation *texec* bridges the gap between the small-step semantics and the big-step semantics. For example, considering the execution of the *out* instruction in Figure 6.2 for the states *nstate s* and *nstate t* to be in *texec*, there need to exist the respective transitions on the level of the small-step semantics. We show the following lemma:

Lemma 8 (Big-step implies texec-steps)
If $instrSet = \cup_{sc}\ strInstr$ ***and*** $instrSet \subseteq lis\ s_{PId}$ ***and***
$((s, trs), {}_{sPId|}strInstr, t, trt) \in$ *big-step* ***then***
$(nstate\ s, trs, instrSet, nstate\ t, trt) \in texec$

It states that given a big-step execution from state s to state t through the structured code *strInstr*, there exists an equivalent execution on the level of the timed small-step semantics that by executing the unstructured set of instructions *instrSet* moves from an *nstate s* to an *nstate t*. The relation between structured and unstructured code is described by the function $\bigcup_{sc}$. It maps a given piece of structured code to the set of labeled instructions it consists of. The proof of the lemma uses rule induction on the big-step relation. The base cases require that every big-step transition implies that the corresponding series of small-step transitions as described by *texec* is possible. The rule for sequential composition requires an induction over the rules of *texec*. It shows that a big-step execution from s to t with respect to the structured code $sc1 \oplus sc2$ is reflected by a *texec* execution from s through the unstructured version of $sc1$ to an intermediate state s' and then from s' through the unstructured version of $sc1 \oplus sc2$ to t (and also the case where execution first moves through $sc2$ and then through the union).

For showing that also every terminating small-step execution has a big-step counterpart, we show a similar lemma. However, here we use an indexed multistep relation *texec_k*, This relation is similar to *texec* in the sense that it 'collects' the intermediate states present on the level of the small-step semantics and describes terminating executions. More precisely, it is equivalent to *texec* in terms of terminating executions, .i.e., for every execution in *texec* there exists a k so that a corresponding execution is in *texec_k* and for every execution in *texec_k* there exists a corresponding execution in *texec*. We use the variable k as a step counter, which facilitates induction over possible executions. Note

that there is not an execution in *texec_k* for every k (because the program counter may still point to an instruction in *instrSet*). The following lemma establishes that when executing the instruction set $\cup_{sc}\ sc$ from a *nstate* s and execution ends in *nstate* t, this execution on the small-step level is reflected by an equivalent big-step transition.

Lemma 9 (Timed small-step implies big-step)
If $wff_{sc}\ sc$ ***and*** $\cup_{sc}\ sc \subseteq lis\ s_{PId}$ ***and***
$(nstate\ s,\ trs,\ \cup_{sc}\ sc,\ nstate\ t,\ trt,\ k) \in texec\text{-}k$ ***then***
$((s,\ trs),\ {}_{sPId|}sc,\ t,\ trt) \in big\text{-}step$

The proof is realized by induction on the type of structured code. For the base cases, a case distinction on the rules defining *texec_k* is required. The induction step then requires an induction on the unstructured set $sc1 \oplus sc2$ regarding possible *texec_k* executions. This induction is realized using the variable k and is the reason for using the indexed multi-step relation *texec_k*. Using the equivalence between *texec* and *texec_k* as explained above, we can establish that there exists a corresponding big-step execution for every *texec* execution.

The lemmas shown in this section enable us to transfer properties established on the big-step level using compositional techniques to the level of the small-step semantics. Furthermore, for given executions from a *nstate* to a *nstate* on the small-step level, we ensure that there exists a related big-step execution.

6.4 Proof Logic for Timed Communicating Low-Level Code

In this section, we present a proof logic that corresponds to the big-step semantics for the low-level real-time language defined in the previous section. The inference rules of the proof calculus can be used to establish specifications about low-level code on an convenient level of abstraction. By establishing the soundness of the provided proof system, it is ensured that information about the behavior of a program can be transferred to the level of the big-step semantics and from there to the level of the timed small-step semantics.

Again, we define assertions following the extensional approach, i.e., we do not define an assertion language explicitly but only fix the type of assertions. Compared to the proof calculus presented in Section 5.4 from the last chapter, the main difference is that assertions now additionally depend on timed communication traces as defined in the last section for the big-step semantics. This leads to the following type of assertions:

type-synonym $('aux,'a,'b)ass = 'aux \Rightarrow (('a)state \times ((real \times ('b)eventplus)\ list)) \Rightarrow bool$

For terminating executions, the logic therefore relates assertions about state-trace pairs with assertions about state-trace pairs. Similarly to the big-

$$\left\langle \begin{array}{l} \lambda\ aux\ s.s_{PC} = l \wedge s_{PId} = p \wedge f\, s_R \neq \emptyset \wedge (\forall x.\ (x,\ l) \notin f\, s_R) \wedge \\ \quad (\forall x\ m.\ (x,\ m) \in f\, s_R \wedge \\ \quad\ (q\ aux\ ((s\ (\!|R := x, PC := m, tnow := t_{up}(s, p, do\ f)|\!)), ts))) \\ \quad \vee\ s_{PC} \neq l \wedge q\ aux\ (s,\ ts) \end{array} \right\rangle\ {}_p|(l_{::} do\ f)\ \langle q \rangle\ \text{HDOF},$$

$$\left\langle \begin{array}{l} \lambda\ aux\ s.s_{PC} = l \wedge s_{PId} = p \wedge tol = 0 \wedge (\exists e.\ f\, s_R = e) \wedge \\ (\forall tx > 0.(q\ aux\ ((s\ (\!|R := x, tnow := t_{up}(s, p, do\ f) + tx|\!)), \\ \quad ts \bullet [(s_{Ptnow} + (tibef\ (out\ f\ fto\ tol)\ p + tx),\ ev\ (f\, s_R))]))) \\ \vee\ s_{PC} \neq l \wedge q\ aux\ (s,\ ts) \end{array} \right\rangle\ {}_p|(l_{::} out\ f\ fto\ tol)\ \langle q \rangle\ \text{HOUT},$$

$$where\ t_{up}(s, pid, instr) := (tnow\ s) + tibef(instr)\ pid + tiaft(instr)pid$$

Figure 6.5: Excerpt from the Rules of the Total Correctness Calculus

step semantics, we adopt the rules of the proof logic to cope with timed communication traces. For the rules dealing with instructions considered to be internal, i.e., non-communication instructions, the extension to the new language is straightforward because the respective instructions do not alter the communication trace.

Figure 6.5 gives the proof rules for the instructions *do f* and *out*. The first is the rule for the instruction *do f*. The structure of the proof rules follows along the lines of the rules for the basic language in Chapter 5. For the instruction *do f* the precondition needs to ensure that, if the program counter points to the instruction and the structured code is indexed with the id of the respective state, the set of states given by f is not empty and that f does not yield a tuple of store and label where the label points to the current label l. If furthermore the postcondition q holds for every pair of store and label given by f, then the postcondition q also holds after execution of the instruction *do f*. If the program counter does not point to the instruction and the postcondition holds, then it also holds. The rule for the instruction *out* is different from the previous one because now the communication trace is updated. If no timeout is specified (because $tol = 0$), then the communication of the output event can occur at any time tx after the preparation time of the instruction ($tibef(out\ f\ fto\ tol)$). Therefore the event $ev\ (f\, s_R)$ is appended to the timed trace ts with a timestamp obtained by adding the preparation time and the waiting time tx to the time value of the starting state s. The rules for the other instructions are adjusted in a similar way.

The rules of our proof calculus for real-time low-level code with communication primitives are sound with respect to the operational semantics. The termination predicate and the notion of validity are defined as in Chapter 5.

Theorem 5 (Soundness for total correctness of real-time code)
$\vdash_t \langle p \rangle\ code\ \langle q \rangle \longrightarrow \models_t \langle p \rangle\ code\ \langle q \rangle$

$$\frac{(s,\ e,\ t) \in tsmallstep}{(node\ s,\ e,\ node\ t) \in ptsmallstep}\text{SSTEP}$$

$$\frac{e \in netChs\ s \cap netChs\ s' \qquad (s,\ ev\ e,\ t) \in ptsmallstep \qquad (s',\ ev\ e,\ t') \in ptsmallstep}{(cluster\ s\ s',\ ev\ e,\ cluster\ t\ t') \in ptsmallstep}\text{PSTEP}$$

$$\frac{\forall\, ta.\ e \neq time\ ta \qquad (\exists\, e'.\ e = ev\ e' \wedge e' \notin netChs\ s \cap netChs\ s') \vee e = \tau \qquad (s,\ e,\ t) \in ptsmallstep \qquad s' = t'}{(cluster\ s\ s',\ e,\ cluster\ t\ t') \in ptsmallstep}\text{PISTEP1}$$

$$\frac{\forall\, ta.\ e \neq time\ ta \qquad (\exists\, e'.\ e = ev\ e' \wedge e' \notin netChs\ s \cap netChs\ s') \vee e = \tau \qquad s = t \qquad (s',\ e,\ t') \in ptsmallstep}{(cluster\ s\ s',\ e,\ cluster\ t\ t') \in ptsmallstep}\text{PISTEP2}$$

$$\frac{0 < d \qquad (s,\ time\ d,\ t) \in ptsmallstep \qquad (s',\ time\ d,\ t') \in ptsmallstep}{(cluster\ s\ s',\ time\ d,\ cluster\ t\ t') \in ptsmallstep}\text{PTSTEP}$$

Figure 6.6: Transition Rules for Networks of Low-Level Components

The soundness theorem is especially important because it allows for the transfer of properties established using the proof calculus to the level of the big-step semantics. Using the theorems from the last section, we can then further transfer the properties to the level of the small-step semantics. In Section 7.2 of Chapter 7 we exploit this strategy to aid the specification and verification of bisimulation relations. We have not yet established completeness for this proof calculus formally, but we expect the proof strategy from the last chapter to succeed here as well.

In the following section, we shortly explain how the stack of semantics presented in this chapter can be extended to a distributed setting.

6.5 Towards Distributed, Communicating, Timed Low-Level Code

Our main goal of obtaining a small-step semantics for timed communicating low-level code and providing a matching big-step semantics and proof calculus for total correctness is realized by the concepts presented in the last sections. In this section, we explain how the presented semantic framework can serve as the basis for a compositional semantics and related proof calculus for distributed low-level languages. We therefore extend the timed small-step semantics of our timed low-level language to a distributed setting. We argue that the language can be given a big-step semantics and related proof calculus, which enable compositional proofs about distributed low-level code.

6.5.1 Networks of Sequential Processes

Based on the semantics for sequential low-level proceses from the last section, we now define a semantics for networks of low-level programs. The syntax of these programs is restricted in the sense that parallelism can only be used on the outer level. Two programs are connected by unidirectional channels, which they exclusively share. A distributed state is then the pairwise combination of the individual component states.

datatype $'a\ net = node\ ('a\ extstate) \mid cluster\ ('a\ net)\ ('a\ net)$

Based on the behavior of individual components, we model the behavior of networks of components using the compositional rules given in Figure 6.6. The rules resemble the behavior of the alphabetized parallel operator known from Timed CSP.

We have established similar lemmas about the timing behavior of the distributed version of our semantics as we have for the sequential version of the semantics, i.e., the passage of time is deterministic, only positive time values are possible and timed transition can be added and interpolated. These lemmas allow us to interpret the distributed semantics in terms of a timed labeled transition system.

To obtain a compositional semantics for the distributed version of our timed low-level language, we follow an approach from [Zwi89] for high-level languages, that was later extended to a real-time setting in [Hoo98a]. The basic idea realized there is that the behavior of a network of distributed components, which share no variables, can be obtained by merging their communication traces. However, we have not fully developed this semantics, but based on our current results we are confident that our compositional semantics for sequential timed low-level code can serve as a basis for such a semantics and a corresponding proof calculus.

6.6 Summary

In this chapter, we have presented a timed low-level language with communication primitives. We have first focused on the timed communication behavior of sequential programs by defining a small-step semantics. Since the communication and the timing behavior of such programs is reflected in terms of labeled transitions in the definition of this semantics, it can be interpreted as defining a timed labeled transition system. This representation serves as the basis for our relation between low-level code and process-algebraic specifications presented in the following chapter. The timed labeled small-step semantics of our language is quite complex. This is due to the states required to realize the timing and communication behavior, which facilitate the interpretation of the semantics as defining a labeled transition system. To reduce the complexity

of analyzing the behavior of programs in our extended low-level language, we have defined a big-step semantics and related proof logic. By having established that the big-step semantics agrees with the small-step semantics and furthermore that our proof logic is sound, we are able to transfer properties established on the level of the proof calculus to the level of the timed labeled small-step semantics.

Using the concepts mentioned above, we have laid the cornerstone for our main goal: relating sequential low-level programs to their abstract specifications given in a process-algebraic representation. Furthermore, we have briefly explained that our semantics can be used as the basis for the analysis of networks of low-level programs as well. To this end, we have defined a distributed small-step semantics for our timed low-level language.

In the next chapter, we build on the concepts from this chapter and explain how the notion of bisimulation can be used to relate process-algebraic specifications and low-level code. Furthermore, we explain how the proof logic for timed sequential low-level code from this chapter can be used to discharge proof obligations in bisimulation proofs, thereby lifting bisimulation proofs to a more abstract level.

7 Proofs about Conformance

In the last chapter, we have defined a semantic stack for the analysis of timed communicating low-level code. Using this stack of semantics, we are able to analyze low-level code on different levels of abstraction, i.e., on the level of the small-step semantics or on the level of the big-step semantics using our proof logic. Furthermore, we have shown that behaviors on the different levels correspond to each other. This enables the transfer of verification results from the abstract level down to the most concrete level of the small-step semantics. This is especially useful because we can use the rules of the proof calculus to establish properties about low-level code in a structured and property-oriented way and also transfer these properties to the most concrete semantic level.

In this chapter, we present a formal framework which enables us to formally relate given instances of low-level code to their process-algebraic specifications. We present a unified semantics and explain how it can be interpreted as a timed labeled transition system. Then we use the notion of weak timed bisimulation to enable mechanized conformance proofs between low-level code and process-algebraic specifications. To cope with the complexity of such proofs, we show how our proof calculus from the previous section can be used to support such conformance proofs. Furthermore, we explain how properties established using our proof calculus can be used for the specification and verification of conformance relations.

Using this approach for the specification and verification of conformance relations between process-algebraic specifications and their low-level implementations, we are able to ensure that properties established on the basis of the process-algebraic specifications can be transferred to the low-level implementation.

7.1 Bisimulations in Isabelle/HOL

In general, the notion of bisimulation is defined based on the concept of labeled transition systems. The intuition is that two states in a labeled transition system can be considered *bisimilar*, if only equivalent behaviors are possible from them. While strong bisimulation requires that every transition possible from one of the bisimilar states is also possible from the other state, weak bisimula-

$$\frac{\begin{array}{l}(P, Q) \in X \\ (\forall (P1, Q1) \in X. \\ \quad \forall\, e\ P2.(P1, e, P2) \in T \rightarrow (\exists\, Q2.(Q1, e, Q2) \in \hat{\mathrm{T}}\ \wedge\ (P2, Q2) \in X \cup \mathit{bisimilar}) \\ \quad \forall\, e\ Q2.(Q1, e, Q2) \in T \rightarrow (\exists\, P2.(P1, e, P2) \in \hat{\mathrm{T}}\ \wedge\ (P2, Q2) \in X \cup \mathit{bisimilar}))\end{array}}{(P, Q) \in \mathit{bisimilar}}$$

Figure 7.1: Proof Principle for Bisimulations

tion abstracts from possible internal transitions. Therefore, weak bisimulation requires that a possible transition from one of the states is also possible from the related state, but arbitrarily many internal transitions are allowed before and after it. Weak timed bisimulation then also considers timed transitions, i.e., it is defined based on the notion of a timed labeled transition system. For two states to be weakly timed bisimilar it is then required that possible timed transitions originating from one of the states can be identified with a number of timed transitions for which the sum of the durations of the individual transitions equals the overall duration. The individual timed transitions are furthermore allowed to be interleaved by an arbitrary amount of internal transitions.

In Isabelle/HOL, we define the notion of bisimulation as introduced in Section 2.3.2 following [Göt12]. Bisimulations are defined as a coinductive set using the directive `coinductive_set` as follows:

```
coinductive_set bisimilar::('s,'a)lts ⇒ ('s,'a)lts ⇒ ('s×'s)set
for T:: ('s,'a)lts and T̂ :: ('s,'a)lts where

⟦ ∀ e P2. (P1,e,P2) ∈ T ⟶ (∃ Q2.
          (Q1,e,Q2) ∈ T̂ ∧ (P2,Q2) ∈ bisimilar ∧
  ∀ e Q2. (Q1,e,Q2) ∈ T ⟶ (∃ P2.
          (P1,e,P2) ∈ T̂ ∧ (P2,Q2) ∈ bisimilar ⟧

⟹ (P1,Q1) ∈ bisimilar
```

Being the dual definition to inductively defined sets, this definition is interpreted in terms of largest fixpoints, i.e., the rules defined above define that a given relation is a bisimulation relation if it is closed under the specified rules and represents the largest fixpoint. From this definition, the Isabelle/HOL system automatically derives a proof principle for proving concrete relations to be bisimulations. This is given by the inference rule in Figure 7.1.

We use this principle for concrete bismulation proofs between process-algebraic specifications and their low-level implementations later in this chapter. To apply it in concrete situations, the set X is instantiated with a bisimulation relation. In our setting, such a relation relates process-algebraic processes with low-level states. The low-level states are interpreted with respect to a given low-level program. For a given tuple (P, Q), to establish that bisimilar behaviors are possible, we first need to show that the tuple is in the specified relation X. Then, it needs to be shown that for every tuple of the relation, if the process performs a step, this step can be answered by a step of the low-level program that again yields a tuple that is in the relation X.

$$\frac{s\ -<event>\rightarrow\ t}{(both.tcsp\ s,\ event,\ both.tcsp\ t)\ \in\ OpSemBoth}\text{TCSP}$$

$$\frac{(s,\ event,\ t)\ \in\ tsmallstep}{(ll\ s,\ event,\ ll\ t)\ \in\ OpSemBoth}\text{LLVM}$$

Figure 7.2: Rules of the Combined Semantics

The definition of bisimulation as given above is defined on a single transition system T. To make the definition applicable in our setting, we therefore unify the operational semantics of our low-level language with the operational semantics of Timed CSP in order to obtain a semantics, which in turn can be interpreted as defining such a transition system T.

7.1.1 A Unified Timed Labeled Transition System

In the last chapter, we have constructed a semantics for a real-time low-level language that can already be interpreted as defining a timed labeled transition system. To use the notion of weak-timed bisimulation to relate low-level code and process-algebraic specifications, we need to relate states of the low-level semantics, given by a notion of state *(('a)extstate))* to states of the process-algebra given as process descriptions *(('v,'b)Process)*[1] (where $'v$ refers to a set of process names and $'a$ to an event alphabet). We therefore integrate the TLTS semantics of the two involved formalisms into a unified transition system (or semantics). We formalize this by defining a unified state *('v,'a,'b)both*. Using the constructors *ll* and *tcsp*, we separate the combined state into low-level program states and Timed CSP process states:

datatype $('v,\ 'a,\ 'b)\ both = ll\ ('a)\ extstate \mid tcsp\ ('v,\ 'b)Process$

Based on this combined notion of state, we define the possible transitions in the new TLTS based on the operational semantics of the timed low-level language on the one side and the operational semantics of Timed CSP on the other side: if there is a transition possible in the individual semantics there is a transition in the unified transition system. The semantics is defined using the rules given in Figure 7.2. The notation $s\ -<event>\rightarrow\ t$ describes a possible transition of a Timed CSP process. In the definition of the rules, *event* refers to both timed and event transitions[2].

As it is our goal to interpret the unified semantics as a timed labeled transition system, we verify the necessary properties. The interpretation as a timed labeled transition system will later allow us to use the definitions of bisimulations for concrete conformance proofs.

[1] This type of process descriptions *(('v,'b)Process)* is defined in the formalization of the semantics of Timed CSP given in [Göt12].

[2] The identifier *both* used here and in further definitions is used by the Isabelle/HOL system to avoid naming conflicts.

The first lemma states that if there exists a time step in the unified semantics, it is either a step of the low-level semantics with a positive time value or it is a Timed CSP step with a positive value.:

Lemma 10 (Time Steps Unified)
If $(P,\ time\ t,\ Q) \in OpSemBoth$ ***then***
$\exists\, p\ q.\ P = both.tcsp\ p \wedge Q = both.tcsp\ q \wedge p\ -<time\ t>\rightarrow\ q \wedge 0 < t \vee$
$\exists\, p\ q.\ P = ll\ p \wedge Q = ll\ q \wedge (p,\ time\ t,\ q) \in tsmallstep \wedge 0 < t$

As also established for our timed small-step semantics in Section 6.2, the unified semantics has the property of being deterministic with respect to the passage of time:

Lemma 11 (Time Determinsim Unified)
If $(P,\ time\ t,\ P1) \in OpSemBoth$ ***and*** $(P,\ time\ t,\ P2) \in OpSemBoth$ ***then*** $P1 = P2$

The values of individual time steps can be added up and there exists a timed step for the resulting value, i.e., there is a transition from $P1$ to $P3$ that takes the time it takes from $P1$ to $P2$ and from $P2$ to $P3$.

Lemma 12 (Time Additivity Unified)
If $(P1,\ time\ t1,\ P2) \in OpSemBoth$ ***and*** $(P2,\ time\ t2,\ P3) \in OpSemBoth$ ***then*** $(P1,\ time\ (t1 + t2),\ P3) \in OpSemBoth$

The following lemma can be considered to be the converse of the lemma for the addition of time steps. It states that in the unified semantics, any time steps can be split into two individual time steps and that there exists an intermediate state $P2$.

Lemma 13 (Time Interpolation Unified)
If $(P1,\ time\ (t1 + t2),\ P3) \in OpSemBoth$ ***and*** $0 < t1$ ***and*** $0 < t2$ ***then***
$\exists\, P2.\ (P1,\ time\ t1,\ P2) \in OpSemBoth \wedge (P2,\ time\ t2,\ P3) \in OpSemBoth$

Now that we have integrated the semantics of Timed CSP processes and low-level programs into a unified semantics and established the properties which allow for the interpretation of the unified semantics as a timed labeled transition system, we can instantiate the abstract definition of a timed labeled transition system. Technically, this is achieved by establishing that the context of our newly defined TLTS satisfies the abstract requirements fixed within the definition of the locale `timed_lts`. This locale fixes exactly the assumptions that correspond to the lemmas proved above.

7.1.2 Example

In this section, we use a small example to explain our verification strategy for bisimulations between Timed CSP specifications and low-level implementations on the level of the timed small-step semantics. Even though we only

use a small program fragment consisting of only one instruction, the following proof of weak bisimilarity already shows the complexity of proofs on the level of the small-step semantics. Recall that in the last chapter in Section 6.2, we modeled the execution of instructions as consisting of three parts: the amount of time that might pass by before an instruction is executed, the execution of the instruction in terms of its externally visible behavior combined with the effect on the internal state and, as the last part, the time that passes by after the instruction. The behaviors involved on the level of the small-step semantics when executing an instruction are therefore more complex than just changing the value of an internal variable. This is a main motivation for the abstraction theorems presented in the next section, which allow us to transfer analysis results from the higher level of abstraction of the proof calculus to the level of the timed-small step semantics. On this higher level of abstraction, the complex transitions mentioned above are subsumed by only one transition that updates the state and the local clock.

Specification and Implementation

We start with a program that consists of just one `out` instruction. The labeled instruction set *lis* for the process id 1 is thus defined as {(1,out outstart yieldZero 0)}. The function *outstart* yields the event $(thr1, start)$ for any given state. Here, $thr1$ is supposed to be an identifier for a thread, for example, and *start* is used to signal to the environment that the thread $thr1$ starts its operations. The function *yieldZero*, which is evaluated with respect to the state of the program always yields 0. Therefore, no timeout is given and the communication of the event can be arbitrarily delayed. As no timeout is given, the timeout label is set to 0. Recall that the semantics is defined polymorphic with respect to the type of events. Here, we have fixed the type of events to consist of tuples where the first part is a thread identifier and the second is a command. Furthermore, we set the *tibef* and *tiaft* values, which specify the amount of time that passes by before and after the visible event of the instruction to the value 1. Technically, this is achieved when instantiating the locale definition for this example.

The specification for the one instruction program directly reflects the labeled transition semantics of the respective instruction as described above:

$$Thread1 = WAIT(1);\ (thr1, start) \rightarrow WAIT(1);\ STOP$$

The specification expresses that one time unit passes by. Afterwards, the process can synchronize on the event $(thr1, start)$. If the environment is ready to perform $(thr1, start)$, the process lets one time unit pass by and afterwards does nothing but letting time pass by.

Definition of the Bisimulation Relation

We define the bisimulation relation as the union of individual relations. The individual relations identify the states from which equivalent behavior is possible. Conceptually, two kinds of relations can be distinguished:

- **Control state relations** relate states from which either communications are possible or time may pass by. In the latter case, a time step leads to a state in the following relation.
- **Intermediate state relations** relate time states that connect control states. In these relations, we use a variable to identify the infinitely many substates that are possible when time passes by between two control states.

For the process definition mentioned above and the instruction that implements the specification, we define the bisimulation relation as follows. The first relation is a control state relation and identifies the process specification (denoted in the definition by P) with the initial state s that the implementation is started in. The assertion requires that s has the process Id 1 and that the program counter in s points to the label one.

$$
\begin{aligned}
&\mathit{bisimRel1} = \{\ (P,Q).\ \exists\ s\ .\\
&P = (\mathit{tcsp}\ (\mathit{WAIT}\ 1;;(\mathit{thr}\ 1,\ \mathit{start}) \rightarrow \mathit{WAIT}(1);;\mathit{STOP})) \wedge\\
&Q = (\mathit{ll}\ (\mathit{nstate}\ s)) \wedge (\lambda\ s.\ \mathit{PId}\ s = 1 \wedge \mathit{PC}\ s = 1 \wedge \mathit{tnow}\ s = 0)\ s\}
\end{aligned}
$$

The second relation identifies the infinitely many intermediate states that are reachable from the states identified in *bisimRel*1. These are states that reflect the passage of time. Here, we use the variable x to denote that exactly one unit of time can pass by. From the second tuple, the specified process finishes the *WAIT* process by taking an internal transition (due to the combination of *SKIP* and sequential composition). In this tuple, the implementation has reached a state, where the countdown value in the *tbstate* has reached 0 and can now take a transition to the *evdelstate*.

$$
\begin{aligned}
&\mathit{bisimRel2a} = \{\ (P,Q).\exists\ s\ x.\ (x \geqslant 0 \wedge x < 1 \wedge\\
&P = (\mathit{tcsp}\ (\mathit{WAIT}(x);;(\mathit{thr}\ 1,\mathit{start}) \rightarrow \mathit{WAIT}(1);;\mathit{STOP})) \wedge\\
&Q = (\mathit{ll}\ (\mathit{tbstate}(s,x))) \wedge (\lambda\ s.\ \mathit{PId}\ s = 1 \wedge \mathit{PC}\ s = 1)\ s) \vee\\
&\\
&P = (\mathit{tcsp}\ (\mathit{SKIP};;(\mathit{thr}\ 1,\mathit{start}) \rightarrow \mathit{WAIT}(1);;\mathit{STOP})) \wedge\\
&Q = (\mathit{ll}\ (\mathit{tbstate}(s,0))) \wedge (\lambda\ s.\ \mathit{PId}\ s = 1 \wedge \mathit{PC}\ s = 1)\ s\}
\end{aligned}
$$

In the next relation *bisimRel*2*b*, states of the specification and of the implementation are identified from which time can arbitrarily pass by until the communication takes place. Although time can pass by from the identified states, we consider these states to be control states. This is because a visible communication is possible here after an arbitrary amount of time passes by. However, we do not have to model the passage of time here explicitly.

$$
\begin{aligned}
&\mathit{bisimRel2b} = \{\ (P,Q).\ \exists\ s.\\
&P = (\mathit{tcsp}\ ((\mathit{thr}\ 1,\mathit{start}) \rightarrow \mathit{WAIT}(1);;\mathit{STOP})) \wedge\\
&Q = (\mathit{ll}\ ((\mathit{evdelstate}\ s))) \wedge\ (\lambda\ s.\ \mathit{PId}\ s = 1 \wedge \mathit{PC}\ s = 1)\ s\}
\end{aligned}
$$

After communication takes place, both the specification and the implementation have the possibility of letting exactly one time unit pass by. The specified relation *bisimRel2c* is similar to *bisimRel2a*.

$$\begin{array}{l} bisimRel2c = \{\ (P,Q).\ \exists\ s\ x.\ (x \geqslant 0 \wedge x \leqslant 1 \wedge \\ P = (tcsp\ (WAIT(x);\ ;\ STOP)) \wedge \\ Q = (ll\ ((tastate(s,x)))) \wedge (\lambda\ s.\ PId\ s = 1 \wedge PC\ s = 2)\ s) \\ \vee \\ P = (tcsp\ (SKIP;\ ;\ STOP)) \wedge \\ Q = (ll\ (nstate\ s)) \wedge (\lambda\ s.\ PId\ s = 1 \wedge PC\ s = 2)\ s\} \end{array}$$

After one time unit has passed by the execution of the `out` instruction is finished and the next control state is reached. In this example, the Timed CSP specification stops, i.e., no successful termination is signaled. Similarly, the implementation stops by incrementing the program counter to two, which is outside of the instruction set *lis*. Therefore, the implementation takes an internal transition to the termination state.

$$\begin{array}{l} bisimRel3 = \{(P,Q).\ \exists\ s. \\ P = (tcsp\ STOP) \wedge \\ Q = (ll\ (tsstate\ s)) \wedge\ \ (\lambda\ s.\ PId\ s = 1 \wedge PC\ s = 2)\ s\} \end{array}$$

The final bisimulation relation is then the union of the individual control state and the intermediate state relations as mentioned above. The advantage of this style of specification is that we can separate and reuse parts of specified relations in a modular way. Technically, this is especially useful when dealing with big relations in Isabelle/HOL.

$$bisimRel = bisimRel1 \cup bisimRel2a \cup bisimRel2b\ \cup bisimRel2c\ \cup bisimRel3$$

We now establish that the process *Thread1* is weakly timed bisimilar to the program with the process identifier 1 consisting of the `out` instruction started in a state s that has the process Id 1 and where the program counter points to the label 1. The following lemma formalizes this claim. We use the relation *weak-timed-bisimilar*, which refers to the definition of weak timed bisimulation in the locale context *cs*.

$$\begin{array}{l} \textbf{lemma}\ \forall\ s.(\lambda\ s.\ PId\ s = 1 \wedge PC\ s = 1 \wedge tnow\ s = 0)\ s \longrightarrow \\ \qquad (tcsp\ (asg\ Thread1), (ll\ (nstate\ s))) \in cs.weak\text{-}timed\text{-}bisimilar \end{array}$$

To establish this bisimulation lemma, we need to show that from the specified states equivalent behaviors are possible. The proof principle derived from the coinductive definition of weak timed bisimulation can be used exactly for this purpose.

```
apply(rule_tac X="bisimRel" in cs.weak_timed.bisimilar_coinduct)
```

Using this command, we invoke the proof tactic that implements the proof principle shown in Figure 7.1. We instantiate the set X with the specified relation *bisimRel* and then need to show that it satisfies the requirements of the principle. The proof then proceeds by first establishing that the tuple *(tcsp (asg Thread1), (ll (nstate s)))* mentioned in the lemma is in the specified relation. Then, it needs to be shown that from both states only transitions are possible, which can be simulated from the other state and that only pairs

of states can be reached that are again in the specified relation. For the low-level program this implies that for every state specified in the relation, the possible transitions are analyzed using the rules of the small-step semantics. This includes internal transitions. In the next section, we explain how the proof calculus from Chapter 6 can be used to lessen the burden of separately analyzing all the possibilities of the low-level program for internal transitions.

7.2 Abstraction Theorems for Bisimulations

In our concrete setting of applying the notion of weak timed bisimulation, it is often the case that simple transitions in an abstract representation are reflected by a complex series of transitions in the concrete representation. This is due to the way abstraction is realized in the definition of weak timed bisimulation on the one hand and the fact that we relate behaviors specified in a rather abstract formalism to behaviors given in a very concrete representation of an imperative programming language on the other hand. For our verification environment this implies that we have to cope with high numbers of internal transitions on the level of the low-level implementation when establishing conformance.

In this section, we show how the proof calculus for total correctness presented in the last chapter can be used to reduce the efforts required for concrete bisimulation proofs. The main idea is based on the observation that from an outside or process-oriented point of view, instructions of the low-level language that do not realize input or output behavior only contribute to the externally visible (or process-oriented) behavior of a program in terms of time that passes by. Based on this observation, we characterize the program states visited during the execution of a number of internal instructions abstractly. All information that is necessary about these states is the amount of time which has passed by since the last and before the next visible instruction. In Chapter 4, we have given an intuition for our strategy of abstracting from internal transitions in concrete proofs about bisimulations between process-algebraic specifications and low-level code. It is based on the idea that a bisimulation relation can be specified using a Hoare triple. The pre- and the postcondition need to ensure that no events are communicated and furthermore need to be strong enough to obtain the value of time that the execution of the code from a state fulfilling the precondition to a state fulfilling the postcondition takes. We split the corresponding bisimulation relation into three parts. One that mentions the states that fulfill the precondition, one that specifies the intermediate states, and one that describes the states fulfilling the postcondition. In this section, we describe the theoretical aspects of this concept more formally.

The theorems presented in this section can be used to transfer properties obtained from Hoare triples as described above to the level of the small-step semantics. They especially focus on the intermediate states and the possible executions from these intermediate states. As explained above, this is especially useful to abstract from internal steps in concrete bisimulation proofs.

Recall that proving a bisimulation relation for a given tuple (P, s) consists of proof obligations that take two directions. For low-level code this means that:

1. It needs to be shown that in state s the low-level code can answer any step possible for the process-algebraic specification from P.
2. For all possible steps from s it needs to be shown that they can be answered by the process from P

In the remainder of this chapter, we develop abstract theorems that support the specification and verification of bisimulation relations in situations where a series of time and internal steps is possible from P and a corresponding series of internal instructions (that take both time and communicate no visible events) is executed from s. By abstract, we mean that we can use Hoare triples to derive the necessary information for both cases described above. The advantage of this approach is that on the abstract level of the proof calculus, analyzing low-level code in our environment is far more convenient than directly on the level of the small-step semantics. The main reason is that a single internal step on the abstract level corresponds to four steps on the level of the low-level semantics.

To formally describe a series of internal and timed transitions on the level of the small-step semantics for our unified semantics, we use the relation *time-plus*[3]. It is the mechanized version of the relation $\longrightarrow_{wt}$ introduced in Section 2.3.2. If there exists a series of interleaved internal and time steps starting in s and ending in s' taking exactly t time units, we have $(s, \mathit{time}\ t, s') \in \mathit{time\text{-}plus}$. Figure 7.3 shows such an execution. The green line depicts a series of interleaved time and internal steps, which are covered by the relation *time-plus*. The red line symbolizes the related big-step execution. The respective executions build on instructions from *code* (structured in the case of the big-step execution and unstructured in the case of the small-step execution).

Simulation of Time Steps from the Process-Algebraic Specification

The first abstraction theorem covers the situation where we need to show that a time step in the abstract specification can be simulated by the low-level implementation. We assume a Hoare specification ensuring that from a state s fulfilling an assertion *Pre* (the precondition), by executing *code*, a state t that fulfills an assertion *Post* (the postcondition) is reached and that these assertions are sufficient to derive that no visible events are communicated. We can then conclude that there exists an execution to an intermediate state s' for every time value *ta* within the interval *begin* to *end* (where *begin* and *end* are the timestamps fixed by the precondition for s and by the postcondition for t). Moreover, from this state s' there exists an execution to a state fulfilling the postcondition *Post* that takes exactly the remaining amount of time. The first abstraction theorem (Theorem 6) formalizes this intuition. The forth and fifth

[3] In the lemmas and theorems presented in this section, the relation appears as *timed-lts.time-plus OpSemBoth* τ *time*. This name is due to the system of Isabelle/HOL for qualified names.

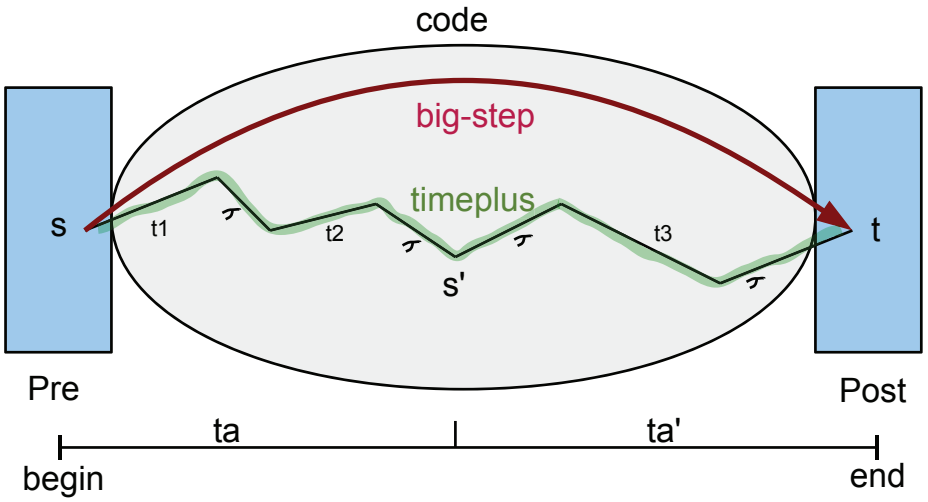

Figure 7.3: Big-step Semantics and Timeplus Executions

premises of the theorem ensure that we can derive from the Hoare specification that no visible events were communicated (using the variable x): the trace has to be the same before and after execution.

Theorem 6 (Abstraction for the Existence of Time Steps)
If $\cup_{sc}\ code \subseteq lis\ p1$ ***and*** $wff_{sc}\ code$ ***and***
$\vdash_t \langle Pre\rangle\ {}_{p1}|code\ \langle Post\rangle$ ***and***
$\forall\, s\ str.\ Pre\ aux\ (s,\ str) \longrightarrow s_{Ptnow} = begin \wedge str = x$ ***and***
$\forall\, t\ ttr.\ Post\ aux\ (t,\ ttr) \longrightarrow t_{Ptnow} = end \wedge ttr = x$ ***and***
$Pre\ aux\ (s,\ str)$ ***and*** $s_{PC} \in_{sc} code$ ***and*** $s_{PId} = p1$ ***and***
$0 < ta$ ***and*** $ta < end - begin$ ***and*** $0 < ta'$ ***and*** $ta' = end - begin - ta$
then
$\exists\ s'\ aux\ t\ tr.$
$(ll\ (nstate\ s),\ time\ ta,\ ll\ s') \in timed\text{-}lts.time\text{-}plus\ OpSemBoth\ \tau\ time \wedge$
$(ll\ s',\ time\ ta',\ ll\ (nstate\ t)) \in timed\text{-}lts.time\text{-}plus\ OpSemBoth\ \tau\ time \wedge$
$(ll\ (nstate\ s),\ time\ (end - begin),\ ll\ (nstate\ t)) \in timed\text{-}lts.time\text{-}plus\ OpSemBoth\ \tau\ time$
$\wedge\ Post\ aux\ (t,\ tr) \wedge tr = x$

The proof of Theorem 6 relies on the relation between the big-step and the small-step semantics. We use the two auxiliary lemmas given below in the proof of the abstraction theorem. First, we establish that there exists a big-step execution for states that fulfill the precondition. Using the Hoare triple, we infer the amount of time that the execution of the respective code takes and furthermore that no communication takes place. Based on this information about the given big-step execution, we can establish that there exists a related small-step execution.

The first auxiliary lemma is formalized as follows. Given a state s from where the program pointer points into the set of code *code* (for which we know that executing it from s terminates) it can be established that there must exist an appropriate transition in our big-step semantics. This is expressed by the following lemma:

Lemma 14 (Existence of big-step transitions)
If $\cup_{sc}\ code \subseteq lis\ s_{PId}$ ***and*** $wff_{sc}\ code$ ***and*** $s_{PC} \in_{sc} code$ ***and*** ${}_{sPId|}code{\downarrow}\ (s,\ str)$ ***then***
$\exists\, t\ ttr.\ ((s,\ str),\ {}_{sPId|}code,\ t,\ ttr) \in big\text{-}step$

The proof of this lemma relies on the definition of the termination predicate ($\downarrow$). It ensures that for a state s the execution of *code* terminates, i.e., that there are no blocking states or that there are no jumps to the own label of an instruction, for example.

Given the existence of such a big-step execution of *code*, starting in a state s that fulfills the precondition *Pre*, the definition of validity for total correctness specifications implies that a state t, reached after executing *code*, must fulfill the postcondition *Post*. In our abstraction theorem we require the precondition to fix the timestamp *begin* and the postcondition to fix the timestamp *end*. The second auxiliary lemma formalizes that given a big-step execution where the communication trace *str* does not grow, there exists a *time-plus* execution that takes the respective duration (Figure 7.3 gives an intuition for this situation).

Lemma 15 (Big-step and timeplus)
If $s_{PC} \in_{sc} code$ ***and*** $\cup_{sc}\ code \subseteq lis\ s_{PId}$ ***and*** $s \neq t$ ***and*** $((s,\ str),\ {}_{sPId|}code,\ t,\ str) \in big\text{-}step$ ***then***
$(ll\ (nstate\ s),\ time\ (t_{PtNow} - s_{PtNow}),\ ll\ (nstate\ t)) \in timed\text{-}lts.time\text{-}plus\ OpSemBoth\ \tau\ time$

The lemma is based on the correspondence between the big-step and the small-step semantics established using Lemma 8 on page 105. The existence of a big-step execution where the communication trace does not grow implies the existence of a small-step execution with the same property. Exploiting the fact that the communication trace does not grow, it can be established that this implies a related *time-plus* execution. Using the second auxiliary lemma, we can conclude that there exists a *time-plus* execution from s to t that has the duration $end - begin$. Given the existence of a *time-plus* execution that takes exactly the time $end - begin$ and where no communications take place, it can further be concluded that there also exists an intermediate state s' for every $ta < (end - begin)$. This follows from the property that every time step can be interpolated in our unified semantics (see Section 7.1.1) and leads to the desired abstraction theorem (Theorem 6).

The next abstraction theorem (Theorem 7) is used in a bisimulation proof when it is required to establish that possible steps of the abstract specification can be answered by the low-level implementation while being in an intermediate state. The premises indicate that an intermediate state s' is reached from a state s fulfilling the precondition after taking a timed step of duration $ta - ta'$, and there exists a transition of duration ta' to a control state t (fulfilling the postcondition). Then it follows that for every point within the interval ta', there exists a further intermediate step.

Theorem 7 (Intermediate States to Postcondition States)
If $ta' < ta$ ***and*** $0 < ta$ ***and*** $0 < ta'$ ***and*** $0 < ta''$ ***and*** $ta'' < ta'$ ***and***
$(ll\ (nstate\ s),\ time\ (ta - ta'),\ ll\ s') \in time\text{-}plus\ OpSemBoth\ \tau\ time$ ***and***
$(ll\ s',\ time\ ta',\ ll\ (nstate\ t)) \in time\text{-}plus\ OpSemBoth\ \tau\ time$
then

$\exists\, s''.\ (ll\ s',\ time\ ta'',\ ll\ s'') \in timed\text{-}lts.time\text{-}plus\ OpSemBoth\ \tau\ time \wedge$
$(ll\ s'',\ time\ (ta' - ta''),\ ll\ (nstate\ t)) \in timed\text{-}lts.time\text{-}plus\ OpSemBoth\ \tau\ time$

This abstraction theorem directly follows from the fact that time steps can be interpolated in the unified semantics (established by Lemma 13 on Page 114).

Using the two abstraction theorems (Theorem 6 and Theorem 7), we can use a Hoare triple to derive the information required to establish that a time step of the abstract specification can be simulated by the low-level implementation. The assertions of the triple need to ensure that the timing interval is long enough and that no communication takes place in the given interval. This is achieved by requiring the communication trace not to grow while executing the low-level code and captured by the Hoare triple. The existence of a *timeplus* execution follows directly from the existence of a big-step execution.

Possible Low-Level Steps

The two abstraction theorems shown in the previous section can be used in situations where a bisimulation proof requires us to show that a time step can be simulated by a low-level step or a series of such steps. The following abstraction theorems are suited for the converse situation: given a tuple of the bisimulation relation, it needs to be shown that every possible transition of the low-level implementation leads to a state, which is again covered by the bisimulation relation and that it can furthermore be answered by the process-algebraic specification. In this section, we present two abstraction theorems that can be used to discharge such proof obligations for intermediate state as specified using our approach.

To provide these theorems, we need to derive from our Hoare specifications that from a state s fulfilling the precondition, execution can only move to states s' either being a control state already fulfilling the postcondition or being an intermediate state from where execution can only move to states t covered by the postcondition. The situation is depicted in Figure 7.4. More precisely, the theorems can be used to deduce possible executions for every state s' reached from a state s fulfilling the precondition via a series of time and internal transitions, where the sum of the durations of the individual steps ta is less than the duration it takes from s to a state t that fulfills the postcondition. From such an s', execution moves to a state t fulfilling the postcondition and takes the remaining amount of time.

The following abstraction theorem (Theorem 8) formalizes the intuition given above. It can be used to show that from a state fulfilling the precondition *Pre* of a given total correctness specification (which ensures that only internal steps are possible on the way to a state fulfilling the postcondition) only states s' are reachable that are intermediate states, i.e., there exists a sequence leading from these states to a state fulfilling the postcondition.

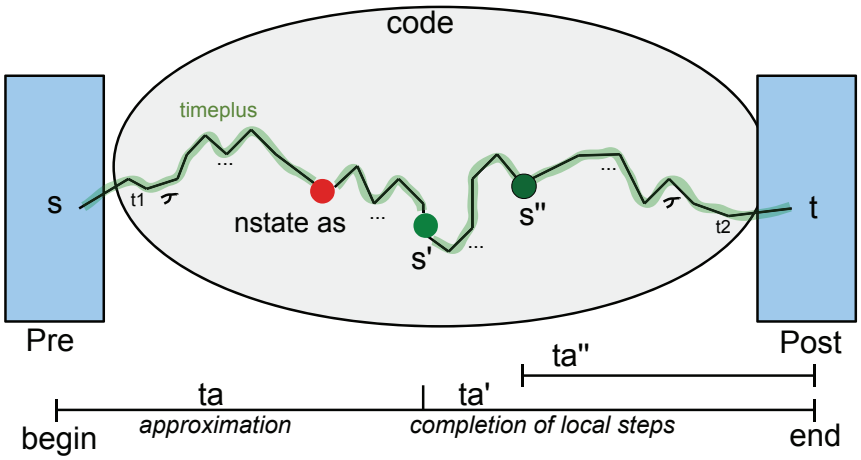

Figure 7.4: Completion of Small-step Executions

Theorem 8 (Possible Steps from Precondition States)
If $\cup_{sc}\ code \subseteq lis\ p1$ ***and*** $wff_{sc}\ code$ ***and***
$\vdash_t \langle Pre\rangle\ {}_{p}1_{|}code\ \langle Post\rangle$ ***and***
$\forall\, s\ str.\ Pre\ aux\ (s,\ str) \longrightarrow s_{Ptnow} = begin \wedge str = x$ ***and***
$\forall\, t\ ttr.\ Post\ aux\ (t,\ ttr) \longrightarrow t_{Ptnow} = end \wedge ttr = x$ ***and***
$Pre\ aux\ (s,\ str)$ ***and*** $s_{PC} \in_{sc} code$ ***and*** $s_{PId} = p1$ ***and***
$(ll\ (nstate\ s),\ tta,\ q) \in OpSemBoth$
then
$\exists\ s'\ t\ ta\ aux\ tr.$
$(q = ll\ s' \wedge tta = time\ ta \wedge 0 < ta \wedge ta < end - begin \wedge$
$(ll\ (nstate\ s),\ time\ ta,\ ll\ s') \in timed\text{-}lts.time\text{-}plus\ OpSemBoth\ \tau\ time \wedge$
$(ll\ s',\ time\ (end - begin - ta),\ ll\ (nstate\ t)) \in timed\text{-}lts.time\text{-}plus\ OpSemBoth\ \tau\ time \wedge$
$Post\ aux\ (t,\ tr) \wedge tr = x$

The theorem given above enables us to deduce which transitions are possible from states fulfilling the precondition. It ensures that only intermediate states s' can be reached from which executions to a state fulfilling the postcondition exist. For these states s', we can ensure that for any step possible from s' we again reach a state that is an intermediate state, i.e., there are executions leading to a state fulfilling the postcondition. This is formalized by the following theorem. It states that given an intermediate state $ll\ s'$ reachable from a state $nstate\ s$ fulfilling the precondition Pre, it can be concluded from the respective total correctness specification that for any possible step to a state q, we can deduce that it is an intermediate state from which a state $nstate\ t$ fulfilling the postcondition is reachable.

Theorem 9 (Possible Steps from Intermediate States)
If $\cup_{sc}\ code \subseteq lis\ p1$ ***and*** $wff_{sc}\ code$ ***and*** $\vdash_t \langle Pre\rangle\ {}_{p}1_{|}code\ \langle Post\rangle$ ***and***
$\forall\, s\ str.\ Pre\ aux\ (s,\ str) \longrightarrow s_{Ptnow} = start \wedge str = x$ ***and***
$\forall\, t\ ttr.\ Post\ aux\ (t,\ ttr) \longrightarrow t_{Ptnow} = end \wedge ttr = x$ ***and***
$Pre\ aux\ (s,\ str)$ ***and*** $s_{PC} \in_{sc} code$ ***and*** $s_{PId} = p1$ ***and***
$0 < ta$ ***and*** $0 < ta'$ ***and*** $ta + ta' = end - start$ ***and***
$(ll\ (nstate\ s),\ time\ ta,\ ll\ s') \in time\text{-}plus\ OpSemBoth\ \tau\ time$ ***and***
$(ll\ s',\ time\ ta',\ ll\ (nstate\ t)) \in time\text{-}plus\ OpSemBoth\ \tau\ time$ ***and***

$(ll\ s',\ tta'',\ q) \in OpSemBoth$
then
$\exists\ s''\ ta''\ aux\ tr.$
$((q = ll\ s'' \wedge tta'' = time\ ta'' \wedge 0 < ta'' \wedge ta'' < ta' \wedge$
$(ll\ s',\ time\ (ta' - ta''),\ ll\ s'') \in timed\text{-}lts.time\text{-}plus\ OpSemBoth\ \tau\ time \wedge$
$(ll\ s'',\ time\ ta'',\ ll\ (nstate\ t)) \in timed\text{-}lts.time\text{-}plus\ OpSemBoth\ \tau\ time)\ \vee$
$(q = ll\ s'' \wedge tta'' = \tau \wedge$
$(ll\ s',\ \tau,\ ll\ s'') \in timed\text{-}lts.time\text{-}plus\ OpSemBoth\ \tau\ time \wedge$
$(ll\ s'',\ time\ ta',\ ll\ (nstate\ t)) \in timed\text{-}lts.time\text{-}plus\ OpSemBoth\ \tau\ time)) \wedge$
$Post\ aux\ (t,\ tr) \wedge tr = x$

The proof of these two theorems in Isabelle/HOL is quite technical and requires a lot of auxiliary lemmas about executions with respect to the relations involved. We describe the main steps of the proof here informally to give an intuition why the theorem holds. The theorems require the transfer of the termination property from the level of the big-step semantics to the level of the small-step semantics. This rules out stuck executions and is particularly important because in our non-deterministic scenario (due to the instruction *do f*), it is not sufficient for an execution to be a subsequence of a terminating execution in order to conclude that it is always completed to a terminating execution.

For example, for the execution depicted in Figure 7.4, we can conclude that there must exist a next state reachable from s' because we know that the execution up to the state *nstate as* is a subsequence of a terminating execution. We further know that execution from *nstate as* with respect to the set of instructions *code* needs to terminate in a state fulfilling the postcondition. The intermediate state s' is reached by a series of small-steps leading through *nstate as*. From *nstate as*, execution cannot get stuck (due to the termination predicate) and leads to another *nstate*. Therefore, we can deduce that execution from the intermediate state s' also leads to another *nstate*, from which a state fulfilling the postcondition can be reached or for which the postcondition already holds. Besides termination, we also need to assure that the steps from *nstate as* can only result from an internal instruction.

The following auxiliary lemma (Lemma 16) formalizes that any *time-plus* execution leading to an intermediate state s' can be approximated in the sense that there exists an 'approximation state' *nstate as*, which is a normal state and from which a number of local steps lead to the intermediate state s' (see Figure 7.4). The lemma is mainly used to bridge the gap between big-step executions and timed small-step executions. The term local refers to the fact that on the level of the big-step semantics only normal states exist, i.e., states that correspond to *nstates*. Transitions on the big-step level subsume the additional 'local' steps on the small-step level. The approximation lemma allows us to deduce that there exists a small-step execution to a normal state *nstate as* from which only local steps are required to reach the intermediate state s'. For the approximation state, we know that execution needs to terminate from there, i.e., there exists a next state s''. This also implies that there must be local small-steps from the intermediate state s' to such a s''. This leads to the following lemma:

Lemma 16 (Complete Local Steps)
If $code{\downarrow}s\ (s,\ trs)$ ***and*** $code \subseteq lis\ s_{PId}$ ***and*** $indom\ s_{PC}\ code$ ***and***
$\forall\, t\ trt\ k.\ (nstate\ s,\ trs,\ code,\ nstate\ t,\ trt,\ k) \in texec\text{-}k \longrightarrow t_{Ptnow} = end \wedge trt = trs$ ***and***
$(ll\ (nstate\ s),\ time\ ta,\ ll\ s') \in time\text{-}plus\ OpSemBoth\ \tau\ time$ ***and***
$ta < end - s_{Ptnow}$ ***then***
$\exists\, s''\ ta'''.\ (ll\ s',\ time\ ta''',\ ll\ (nstate\ s'')) \in time\text{-}plus\ OpSemBoth\ \tau\ time \wedge ta''' \leqslant end - s_{Ptnow} - ta$

The premises of the lemma ensure that executions starting in a state s through the instructions in *code* terminate ($\downarrow$) and that the terminating executions take exactly $end - begin$ time units and no communications take place ($trt = trs$). For an execution to an intermediate state s' we then know that there exists local steps leading to a normal state.

Based on the result from Lemma 16 that we can reach a normal state s'' from the intermediate state, we can further deduce that there exists an execution from a normal state s fulfilling the precondition P to the normal state s''. If s'' already fulfills the postcondition, we know that there is an execution from the intermediate state to a state fulfilling the postcondition. If the normal state s'' does not fulfill the postcondition and the program counter still points into *code*, there must be a terminating execution to a state fulfilling the postcondition.

The proof of Lemma 16 is a main step within the proofs of Theorem 8 and Theorem 9. The proof is quite complex because it involves reasoning on the level of the small-step semantics. Especially the amount of work necessary for the inductions over the relation *time-plus* and the relations it is defined from should not be underestimated.

In the next section, we demonstrate the benefits of these abstraction theorems for bisimulation proofs using a small example. It especially demonstrates that when specifying bisimulation relations using our approach, the respective proofs about the relations benefit from the abstraction theorems given in this section by abstracting from concrete small-step executions.

7.2.1 Example

We extend the small example from the previous section. To illustrate how we integrate the abstraction theorems presented above, we insert a section of code that performs some internal operations. This illustrates one of our main motivations mentioned in Chapter 4: to abstract from internal behavior in implementations.

The abstract specification of our process is given as follows:

$$Thread1 = WAIT(1);\ ((thr1, start) \rightarrow WAIT(5);\ STOP) \overset{1}{\triangleright} STOP$$

The abstract specification describes a simple process with a timeout communication. After letting 1 time unit pass by, the process waits for a communication of the event $(thr\ 1, start)$ for 1 time unit. If the environment agrees on the event within the specified timeout, the process lets 5 time units pass by

before stopping. If the environment does not synchronize on the event within 1 time unit, the process stops immediately. The implementation is given by the following set of labeled instructions:

instructionSet = {(1, out outstart yieldOne 5),(2, br 3),(3, br 4)}

The instruction `out` has a specified timeout of 1 time unit (specified by the function *yieldOne*) to communicate the event (*thr* 1, *start*) to the environment (specified by the function *outstart*). If no communication takes place within 1 time unit, the program counter is set to 5, so that the program terminates unsuccessfully (because there is no further instruction to be executed and the PC is not set to 0). If a communication takes place within 1 time unit, execution proceeds with the branching instruction at label 2 that sets the program pointer to label 3. The branching instruction at label 3 realizes another jump to the label 4, where execution stops because no instruction is available to execute.

Definition of the Bisimulation Relation

The following relation *dbisimRel*2*d* describes the part of the bisimulation relation where an event was communicated within the specified timeout interval. Here, the program counter points to label 2 and the related Timed CSP process is capable of letting four time units pass by and then stop.

$$
\begin{array}{l}
dbisimRel2d = \{(P,Q).\ \exists\ s.\\
(P = (tcsp\ (WAIT\ 4\ ;;\ STOP)) \wedge\\
\ Q = (ll\ (nstate\ s)) \wedge (\lambda\ s.\ PId\ s = 1 \wedge PC\ s = 2)\ s)\}
\end{array}
$$

When proving the bisimulation relation, at this point we need to show that time steps of any length between 0 and 4 of the Timed CSP process can be simulated by the low-level program. We use our abstraction Theorem 6 for this purpose. We instantiate the theorem with the following total correctness specification:

lemma *Specification*:

$$
\begin{array}{l}
\vdash_t \langle \lambda\ aux\ (s,tr).\ PId\ s{=}1 \wedge PC\ s{=}2 \wedge tnow\ s{=}TNOW\ aux \wedge tr{=}TR\ aux\rangle\\
\qquad\qquad ({}_{1|}((2_{::}(br\ 3))_{\oplus}(3_{::}(br\ 4))))\\
\quad \langle \lambda\ aux\ (s,tr).\ PId\ s{=}1 \wedge PC\ s{=}4 \wedge tnow\ s = TNOW\ aux{+}4 \wedge tr{=}TR\ aux\rangle
\end{array}
$$

It specifies that from a state where the program counter points to the label 2, it takes 4 time units to reach the state where the program counter points to the label 4. Furthermore, the specification ensures that no visible labels are communicated by fixing the trace in the precondition and ensuring that the possible trace in a state fulfilling the postcondition is the same. Note that we use the auxiliary state *aux* here in order to refer to the values of *tnow s* and *tr* from the precondition in the postcondition. The auxiliary state is defined as a record that consists of the two fields *TNOW* and *TR*. As explained in the last section, the abstraction theorem ensures that there is an intermediate state for every point of time between the starting time and the end time.

However, proving the bisimulation relation also requires to show that possible steps of the low-level program can be simulated by the Timed CSP specification. To deduce the possible steps from a low-level state covered by the relation *dbisimRel2d*, we use Theorem 8. It states that only internal and time steps are possible from such a state.

In both of the described situations, it needs to be ensured that the reached states are again in the bisimulation relation, together with a related Timed CSP process. This is ensured in both theorems, because only intermediate states are reachable from which further transitions to states that fulfill the postcondition of the total correctness specification given above are possible.

These intermediate states are covered by the following relation:

$$
\begin{aligned}
&\mathit{dbisimRel2e} = \{(P,Q).\ \exists\ x\ s\ s'\ t\ .\\
&((x > 0 \wedge x < 4 \wedge\\
&\ P = (\mathit{tcsp}\ (\mathit{WAIT}(x);;\mathit{STOP})) \wedge\\
&\ Q = (\mathit{ll}\ s') \wedge\ (\mathit{ll}\ (\mathit{nstate}\ s),\ \mathit{time}\ (4-x),\mathit{ll}\ \ s') \in \mathit{cs.time\text{-}plus} \wedge\\
&\qquad\qquad (\mathit{ll}\ s',\ \mathit{time}\ x,\ \mathit{ll}\ (\mathit{tsstate}\ t)) \in \mathit{cs.time\text{-}plus}) \wedge\\
&\qquad\qquad (\lambda\ s.\ \mathit{PId}\ s = 1 \wedge \mathit{PC}\ s = \mathit{Suc}\ 1)\ s\ \wedge\\
&\qquad\qquad (\lambda\ s.\ \mathit{PId}\ s = 1 \wedge \mathit{PC}\ s = 4)\ t)\\
&\vee\\
&(P = (\mathit{tcsp}\ (\mathit{WAIT}(0);;\mathit{STOP})) \wedge\\
&\ Q = (\mathit{ll}\ (\mathit{nstate}\ t)) \wedge (\lambda\ s.\ \mathit{PId}\ s = 1 \wedge \mathit{PC}\ s = 4\ )\ t)\\
&\vee\\
&(P = (\mathit{tcsp}\ (\mathit{SKIP};;\mathit{STOP})) \wedge\\
&\ Q = (\mathit{ll}\ (\mathit{tsstate}\ t)) \wedge (\lambda\ s.\ \mathit{PId}\ s = 1 \wedge \mathit{PC}\ s = 4)\ t)\}
\end{aligned}
$$

The relation *dbisimRel2e* describes all the possible states between the state from which the internal behavior at label 2 starts and the state, where the program counter points to the label 4 and no instruction can be executed anymore. The intermediate states are characterized by the fact that they can be reached through an execution that consists of only internal and timed transitions and that from these states, it is again possible to reach a state where the program counter points to the label 4. Note that this characterization only allows executions, where the amount of time taken to reach the intermediate state added to the amount of time taken to reach the final state from the intermediate state equals the specified timing interval (which is 4 time units in this example). To deduce the possible behaviors from the intermediate states described by the relation *dbisimRel2e*, we use Theorem 7 and Theorem 9. The first one ensures that any arbitrary small time steps from the Timed CSP specification can be simulated, while the second is used to show that from intermediate states only internal or time steps are possible that either lead to a further intermediate state or to a state that fulfills the postcondition of the total correctness specification.

Using our specified bisimulation relation, we can now verify that the specification *Thread*1 and the low-level program with process id *PId* 1 started in a state where the program pointer points to 1 indeed show the same behaviors.

lemma $\forall\ s.(\lambda\ s.\ \mathit{PId}\ s = 1 \wedge \mathit{PC}\ s = \mathit{Suc}\ 0 \wedge \mathit{tnow}\ s = 0)\ s \longrightarrow$
$(\mathit{tcsp}\ (\mathit{asg}\ \mathit{Thread1}),\ (\mathit{ll}\ (\mathit{nstate}\ s))\ ) \in \mathit{cs.weak\text{-}timed\text{-}bisimilar}$

In this section, we have explained our approach for the specification and verification of bisimulation relations using a small example. We have demonstrated that using our abstraction theorems, we can prove a bisimulation relation correct without considering the actual small-step transitions. As a prerequisite, we require a Hoare triple assuring that no visible events occur for the respective execution and allowing us to infer the amount of time the execution takes. Given this information, we can treat a complex series of internal steps as if it was just one transition when proving the bisimulation relation.

7.3 Summary

In this chapter, we have presented the part of our verification environment that enables the application of the formal notion of bisimulation in our setting. To reach this goal, we have integrated the semantics of our target formalisms: the process algebra Timed CSP and our non-deterministic real-time low-level language. Based on the timed labeled transition system defined by the integrated semantics, we can make use of the proof principle induced by the coinductive definition of weak timed bisimulation. Using a small example, we have shown how we specify and verify such relations. As an important proof technique, we have explained how we can transfer properties about low-level code established using the proof logic from the last chapter in order to realize bisimulation proofs in a more compact and abstract way. More precisely, using our abstraction theorems, we are able to treat complex series of internal transitions as if they were just one time step. As a premise for the application of our abstraction theorems, we need to prove a Hoare triple using our proof calculus presented in Chapter 6. However, reasoning about the timed behavior on the level of the proof calculus is far more convenient than on the level of our timed small-step semantics because a single internal transition on the higher level of abstraction subsumes a series of transitions on the lower level.

In the next chapter, we take a step away from low-level languages and focus on the level of abstract specifications in CSP. We propose a specification approach for adaptive systems in CSP that facilitates the verification of important adaption properties in the setting of distributed adaptive systems.

8 Distributed Adaptive Systems in CSP

In the last chapters, we have focused on low-level code and investigated the notion of weak timed bisimulation as a conformance relation. Using these concepts, we are able to transfer properties established on the abstract level to the low-level implementation.

In this chapter, we focus on the abstract level again and discuss how so-called adaptive systems can be modeled and analyzed using CSP. We consider a notion of adaptivity that regards adaptive systems as being a subclass of reactive systems. Adaptivity here means that the systems operate in configurations. In response to signals from the environment, the system might find the need to switch from its current configuration to another. We present an approach to specify and analyze adaptive systems with CSP. Together with our approach for the machine-assisted verification of conformance relations, we achieve a mechanized proof environment for the correct construction of adaptive systems from the abstract specification down to the low-level implementation. In this chapter, we use untimed CSP to focus our study on the relation between configurations and the events that might trigger switches between configurations[1]. We exploit the notion of refinement in CSP to realize a specification approach that divides the system specification into different layers, as shown in Figure 8.1. Conceptually, we separate adaption behavior from functional behavior as proposed by Adler et al in [ASSV07]. Thereby, we obtain two specifications. One focusing only on the adaption behavior of the system to be constructed and the other focusing on its functional behavior. The first is refined by a model that captures the implementation details of the adaption processes that the system realizes. This specification is then further refined by a model that adds details about the implementation of functional behavior and that refines the abstract functional specification.

On the adaptive implementation layer, we model systems in terms of components. These components operate in predefined configurations, which are executed dependent on the internal state. Here, the state only consists of adaption relevant information. A components is connected to other compo-

[1] In Chapter 9, we use the style of specification presented here for our case study of a distributed scheduling system and also introduce timing dependencies.

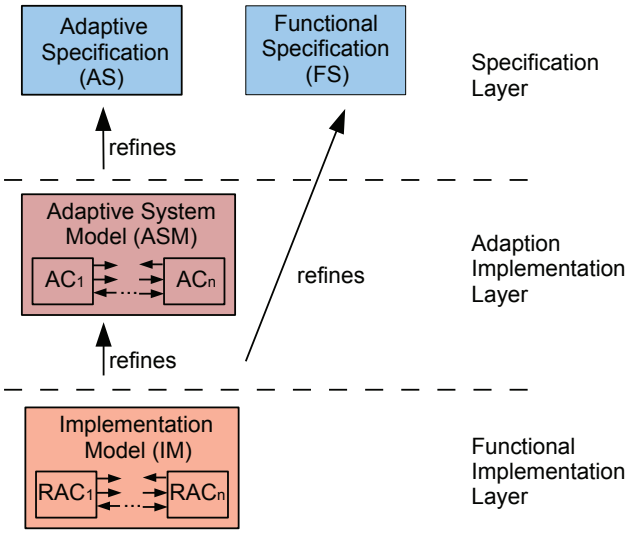

Figure 8.1: Organization of the Specifications

nents by channels over which values can be communicated. These channels are separated into adaptive and non-adaptive channels. The first provide information that might lead to an adaption. The latter are used for communicating information that is not relevant for the adaption behavior of the components. Furthermore, components can communicate with the environment using input and output events. This explicit modeling of adaption actions and information and the strict distinction of functional from adaptive behavior enables us to capture the adaption behavior conveniently in the adaption specification and also enables further analysis. By connecting the communication channels of the components, we obtain the adaptive system model, which can then be analyzed.

8.1 Specification

Classical development approaches using CSP are based on refinement. They start from an abstract specification expressing the system behavior and neglecting implementation details. This abstract specification is then further refined by adding more detailed descriptions of the behavior of the system. In the following, we present a partitioning of specifications that allows us to develop refined models in a uniform manner.

The functional specification (FS) specifies the behavior of the system in terms of input and output behavior that the system is allowed to produce. A functional specification can either be a CSP process, as suggested in Figure 8.1, or an LTL property that is shown to be satisfied by the implementation model of the adaptive system.

Definition 13 (Functional Specification)
A Functional Specification FS specifies the functional behavior of an adaptive system on an abstract level. It abstracts from the adaptive behavior of a system.

In the following, we focus on the specification of adaption behavior. This kind of specification is concerned with the adaption signals that are to be exchanged by the system's components. The functional behavior is concerned with general system inputs and outputs. These sets of channels are not disjoint in general.

8.1.1 Adaptive Specification

On the most abstract level (the topmost layer shown in Figure 8.1), we specify the system's behavior by focusing on adaption behavior and abstracting from implementation details. Configurations and other implementation specific details are not considered on this level. Adaption behavior is modeled in terms of causal dependencies between system input and propagation of adaption signals.

Definition 14 (Adaption Specification)
An Adaption Specification AS specifies the adaption behavior of an adaptive system on an abstract level. It abstracts from functional behavior.

Generally, *ASs* are systems specified as below, in order to partition the adaption specifications in a way suitable for creating adaptive system models and deriving implementations models.

$$AS = |_X|\ i : \{0..n\} \bullet ACS(i)\ .$$

Thus, adaptive specifications are formalized as specifications of adaptive components (ACSs) synchronizing pairwise on adaption events from the set X. The ACSs of the system are formalized using internal (non-deterministic) choice:

$$ACS = \sqcap\ i : \{0..n\} \bullet P(i)\ .$$

In these components, each P specifies the behavior of a particular configuration of the system. If at least one configuration process P of an adaptive component specification terminates, the component is considered to be terminating. If it does not terminate and is deadlock-free, it is resettable and contains a home state,i.e., a state it can always return to. As the definition above suggests, its initial state is likely to be a home state in this latter case.

This style of specification (leaving out implementation details by modeling choices as being non-deterministic) allows us to refine models by adding more

explicit descriptions of internal state to the components and replacing the non-deterministic choices by guarded external (deterministic) choices. As we will see later in this section, this yields component models of the following form

$$CM(s) = \Box\ i : \{0..n\} \bullet g_i(s)\ \&\ P(i).$$

Here, g is a boolean guard on the internal state of the process. Using the notion of CSP refinement introduced in Section 2.4, conformance between the refined models and the specifications for a given initial state s can be established if at least one of the guards always evaluates to true (because of the law 'resolution of non-deterministic choice' $P \sqcap Q \sqsubseteq P$):

$$ACS \sqsubseteq_{\mathcal{FD}} CM(s).$$

$\mathcal{FD}$ denotes refinement in the failure-divergence model here. This way, we obtain adaption models that conform to their specifications by construction.

8.1.2 Adaptive System Model

The adaptive system model specifies the system's behavior in terms of configurations and the behavior of their configuration processes: Each configuration of a process is described by a separate process. Furthermore, interactions with other system components that might trigger an adaption to another configuration within the communicating components are considered on this level.

We model the entities of an adaptive system as a process that offers an external choice over processes that model the behavior of the respective entity in a certain configuration. These processes are guarded by boolean predicates that depend on the internal state s of the component. Changes to the internal state can only be made if the adaptive system engages in an event from the set of adaptive channels AC. Depending on the internal state s and the value x communicated through the channel the component computes the new internal state.

Definition 15 (Adaptive Component)
An Adaptive Component AC_i is defined by an internal state s, a set of guards $G_i = \{g_{i1}, ..., g_{im}\}$, a set of adaptive communication channels $ACC_i = \{ac_{i1}, ..., ac_{in}\}$ and a set of component configurations $CC_i = \{CC_{i1}, ..., CC_{im}\}$. The CCs describe the behavior of the adaptive process in a configuration guarded by predicates $g \in G$ that depend on the components internal state:

$$\begin{aligned} AC_i(s) = {} & ac_{i1}?x \rightarrow AC_i'(f_{i1}(s,x)) \\ & \Box ... \\ & \Box ac_{in}?x \rightarrow AC_i'(f_{in}(s,x)) \end{aligned}$$

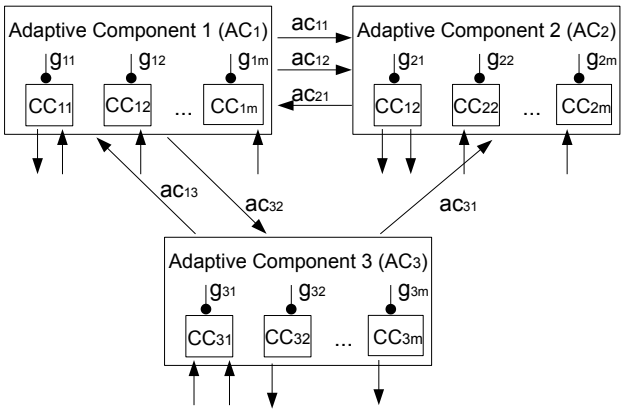

Figure 8.2: Example of an AC Model

$$AC_i'(s) = g_{i1}(s)\ \&\ (CC_{i1};\ AC_i(s))$$
$$\Box ...$$
$$\Box g_{im}(s)\ \&\ (CC_{im};\ AC_i(s))$$

The behavior of a component AC_i in a certain configuration is described by the processes $CC_i = \{CC_{i1}, ..., CC_{im}\}$. These processes handle the input and output behavior of the adaptive component and realize communication with other entities. This means that the control processes signal other processes that adaption is necessary. Note that the control processes do not change the internal state of the adaptive system component.

Definition 16 (Component Configuration)
A Component Configuration CC_{ij} consists of functional input and output events $I_{ij} = \{i_{ij1}, .., i_{ijn}\}$ and $O_{ij} = \{o_{ij1}, ..., o_{ijm}\}$ that realize the system's interaction with the environment. Furthermore, the process uses adaption events $ACP_{ij} = \{ac_{ij1}, ..., ac_{ijn}\}$ with $ACP_{ij} \cap ACC_i = \emptyset$, which signal other components that adaption is necessary.

An adaptive system specification is composed of different ACs by using the parallel composition operator of CSP. A pair of ACs is synchronized on the adaptive and non-adaptive communication channels they share. A further AC can be connected to the system via synchronization on the respective adaptive and non-adaptive communication channels it shares with the combination of the previous two ACs and so on. Using this synchronization strategy, it is possible to realize complex communication networks between the components of an adaptive system. Figure 8.2 shows an example of an AC model.

Definition 17 (Adaptive System Model ASM)
An Adaptive System Model ASM consists of a set $A = \{AC_1, ..., AC_n\}$ of Adaptive Components, which are synchronized pairwise on the communication channels (A_{nm} for AC_n and AC_m) they share. The set A'_{nm} consists of

all events from A_{nm} that are not mentioned in the adaptive specification AS ($A'_{nm} \cap \Sigma_{AS} = \emptyset$)[2]:

$$
\begin{aligned}
ASM = (((&AC_1 \,|_{A_{12}}|\, AC_2) \setminus A'_{12}) \\
&|_{A_{13} \cup A_{23}}|\, AC_3) \setminus A'_{13} \cup A'_{23}) ...) \\
&|_{A_{1n} \cup ... \cup A_{(n-1)n}}|\, AC_n \setminus (A'_{1n} \cup ... \cup A'_{(n-1)n})
\end{aligned}
$$

As a consequence of this construction, the only visible events of the composed adaptive system specification are the input and output events of the individual system components and the adaption events mentioned in the adaption specification.

The adaptive system model can of course be successively refined and analyzed. This is an advantage of our approach that follows directly from the powerful refinement concept in CSP.

8.1.3 Implementation Model

The adaptive system model presented in the previous section focuses on the adaptive behavior of the system. Therefore, the state of the individual adaptive components captures only information that is relevant for the adaption behavior. The control processes for the different configurations are not allowed to change the internal state of the adaptive system component they belong to. The implementation model refines the adaptive system model by refining the states of the individual adaptive components and allowing the configuration processes to modify the internal state of the system components. Functional behavior is specified in an implementation specification. This specification also needs to be refined by the implementation model as introduced in the beginning of this section.

The building blocks of the implementation model are the individual components that refine the adaptive components and collaboratively achieve the behavior specified by the functional specification.

Definition 18 (Refined Adaptive Component RAC)
A Refined Adaptive Component RAC_i extends the internal state s of an AC_i to a state s_f by adding functional variables. The control processes of AC_i are extended by functional behavior and are allowed to modify the extended state s_f.

Refined Adaptive Components are combined like the adaptive components from the previous section and model the behavior of the final system.

[2] Σ_{AS} comprises the entire alphabet of AS

Definition 19 (Implementation Model IM)
An Implementation Model IM consists of refined adaptive system components which are synchronized pairwise on the communication channels they share.

By showing that the system specification refines the adaptive system specification, as explained in the next section, we can prove that the added functionality does not change the adaption behavior.

8.2 Refinement of Adaptive Systems

In the previous section, we have explained our modeling approach for adaptive systems in CSP. The approach presented here assumes the existence of multiple (modular) specifications of a system each focusing on a single aspect. Although these must of course be consistent (in the sense that there exist systems satisfying all of them) we do not require that they use the same alphabets. Thus, to establish a refinement relation between the specifications and the adaptive system and implementation models, hiding and renaming must be used to abstract or transform the alphabets of the models. Only then, conformance between the specifications and the models can be established. This is a standard CSP technique when doing refinement proofs.

We assume that the refined models only add further events to the event sets of the abstract models. However, if the refined models use events with different names, CSP's renaming operator can be used to identify events that model equivalent behavior. By hiding all events from the adaptive system model, which are not present in the adaptive specification, refinement can be achieved if the specification allows the same behavior modulo silent steps.

Definition 20 (Adaptive System Model Conformance)
An Adaptive System Model ASM conforms to an adaption specification AS, iff:

$$AS \sqsubseteq_{\mathcal{FD}} (ASM \setminus (C_{ASM} \setminus C_{AS})),$$

where C_{ASM} and C_{AS} are the unions of all input, output and adaptive events of ASM and AS.

For the implementation model it needs to be shown that it correctly refines the adaptive system model and that it implements the functional specification.

Definition 21 (Implementation Model Refinement)
An Implementation Model IM is a correct refinement of an adaptive system model ASM and of a functional specification FS, iff:

$$ASM \sqsubseteq_{\mathcal{FD}} (IM \setminus (C_{IM} \setminus C_{ASM})),$$

$$FS \sqsubseteq_{\mathcal{FD}} (IM \setminus (C_{IM} \setminus C_{FS})),$$

where C_{ASM} and C_{IM} are the unions of all input, output and adaptive events of ASM and IM.

8.3 Verification

As adaptive systems exhibit a high degree of complexity due to the interaction with their environment and the complex dependencies of their modules, verification of crucial system properties is of superior importance. Besides the adaption and functional properties that are already captured by the adaptive and functional specifications, further adaptive and functional properties of a system can be formally verified in CSP using the automated verification capabilities of FDR and ProB. The CSP-Prover [IR05] also allows us to verify infinite state systems. Our CSP based modeling approach offers the advantage of analyzing the composed system as a whole, as well as analyzing system modules in isolation. In this section, we explain how adaption specific and functional properties can be analyzed.

8.3.1 Stability

For adaptive systems where adaption is not controlled by a central authority, it is important to check whether the system's adaption behavior has the property of stability. Cyclic dependencies between system components may lead to infinite changes of configurations thereby causing the system to perform adaption infinitely often. Adaption behavior is called stable if it is always possible for a system to find a configuration to remain in after performing adaption, given that the input values do not change.

Definition 22 (Stability)
An Adaptive System Model ASM with adaptive communication events from the set A_{ASM} has the property of stability, i.e. is always able to enter a stable state, if it can be proved that:

$ASM \setminus A_{ASM}$ is livelock free.

Freedom from livelock means that the system can not engage in an infinite sequence of silent transitions. Since the adaptive communication events in the definition above become silent transitions, proving livelock freedom ensures that such a chain of silent events is not possible and thus no infinite adaption is possible.

8.3.2 Other Adaption Properties

Besides stability, adaptive systems can exhibit numerous other properties that are of interest. Reachability of configurations, for example, is a desirable property. Having a functional event performed at the start of a configuration process enables us to check for these events in refinement assertions or LTL properties. LTL model checking also enables us to specify fairness properties. The formula

$$\phi = G\ F\ reachableConf$$

is satisfied if and only if the configuration performing the reachableConf event is a home state. This is especially helpful with the alphabet transformer

$$T(P) = P[[x \leftarrow reachableConf \mid x \in ConfEvents]]$$

which renames any event x contained in the set of configuration events *ConfEvents* into *reachableConf*.

8.3.3 General Properties

Further properties of the adaption and the system model can be analyzed using FDR and ProB. For example, checking the composed system for deadlock freedom using FDR ensures that at least one configuration process in every system component is active. Using the model checking capabilities of ProB, crucial system properties of adaptive systems can also be expressed using LTL formulas that express properties about the availability and order of CSP events. As shown by Leuschel et al. [LMC01] and more recently by Lowe [Low08], LTL captures properties different from those expressible via refinement in the standard CSP models (i.e., traces, stable failures, and failures-divergences).

8.4 Discussion

The modeling approach presented in the previous sections is suitable for distributed adaptive systems. A benefit of the proposed way to model adaptive systems is that it allows us to keep track of requirements through the refinement hierarchy (assuming that requirements are not scattered over specifications). In fact, this observation leads to the proposed models, because it separates adaptive behavior from other properties of the system. Dynamic aspects of adaption, for example, the dynamic creation of components and communication channels, cannot be directly modeled using our approach. In CSP, it is not possible to create new communication events or processes on the fly. However, as explained by Roscoe in [Ros10], it is possible to simulate this concept in CSP. If components and channels can be determined at design

time it is thereby possible to simulate dynamic changes using the guards of the adaptive components. For example, the introduction of a new communication channel in a certain configuration can be modeled using a copy of the respective component configuration that might use the new channel. The configuration guards are then used to ensure that the component behaves as before but with a new communication channel available in the configuration that is supposed to use the new channel.

A technical drawback of the approach presented here is performance with FDR. The reason is that FDR is not optimized for dealing with the functional parts of CSP_M that we propose to use heavily on the modeling and implementation level. Optimizing models for FDR involves creating copies of processes by renaming their alphabets in preference to parameterization (as explained in [GRA05], for example). However, ProB and the CSP_M toolkit presented by Fontaine in [Fon10] show no difference in performance to models optimized for use with FDR. Regarding the CSP-Prover [IR05], the models are equally suited for analysis with the theorem proving tool.

8.5 Example

In this section, we present an example that resembles a case study from [SST06]. We model a system that controls the illumination in a room. The light control component that is responsible for the illumination of the room is connected to two sensors. The first one monitors if a room is occupied using a camera or a motion detector. If the camera image is not available, the sensor switches to motion detection and signals its new mode of operation to the control component. The second sensor is a dimmer. If a certain dimming value is set within the component, the light is dimmed if the room is empty. If the value is not set, the light is switched on and off without dimming. Based on the available signals, the light control component adapts its behavior.

Figure 8.3 shows the adaptive specification of the light control system. Note that we use non-deterministic (internal) choice on this level to stress that the resulting process 'SPEC' is a specification. First, we define two communication channels, *camImage* and *dimValue* that represent the input channels of the sensor components. As channels are typed in CSP_M, we assign the predefined type Bool to both channels. The messages communicated over these input channels model the availability of images recorded by the camera and the status of the dimmer. Adaption is modeled by messages communicated over the adaptive communication channels *detect* for the sensor that monitors room occupation, and *lamp* for the sensor that monitors the dimmer. Although both modes are binary in this example, we use the user-defined datatypes *occState* and *lampState*, which model the communication of the detection mode and the switching mode of the lamp, respectively. The intent is to highlight that the adaption messages are clearly causally dependent on an input message, but may depend on other influences. Hence, replaying the original input message as adaption message is not suitable in general. The resulting specification process

```
channel camImage, dimValue : Bool

datatype occState = motion | camera
channel detect : occState
datatype lampState = switch | dim
channel lamp : lampState

CamOff = camImage.false -> detect.motion
                  -> Cam
CamOn  = camImage.true -> detect.camera
                  -> Cam
DimOff = dimValue.false -> lamp.switch
                  -> Dim
DimOn  = dimValue.true -> lamp.dim -> Dim
Cam    = CamOff |~| CamOn
Dim    = DimOff |~| DimOn
SPEC   = Cam    ||| Dim
```

Figure 8.3: Adaptive Specification of the Light Control System

'SPEC' models that both sensors correctly signal environmental changes over their respective adaptive channels.

The adaptive system model that captures the behavior of the implemented adaption mechanisms is shown in Figure 8.4. Following our modeling approach from Section 8.1.2 each component is modeled as an AC process. The AC processes OccSrc and LampSrv, modeling the occupation sensor and the dimmer sensor operate in only one configuration. They receive inputs from the environment through the events camImage?x and dimValue?x. Here, the values of the input signals are modeled as Boolean. Both ACs OccSrv and LampSrv accept the input of the service that they model and propagate the respective adaption messages. For the occupation sensor, these are the values camera and motion through the adaptive channel *detect*. For the dimming sensor, the values are dim and switch through the adaptive channel *lamp*. In both cases the propagated adaption signal depends on the value of the input message x. This behavior corresponds to the understanding that the services operate in cycles reading an input, blocking new inputs while serving a request and then starting over again.

Simple services like the occupancy detection service and the lamp control service can be modeled without an explicit state parameter. Here, the adaption processes within the AC processes depend only on their respective inputs, but not on the state of the AC. The behavior of the light control component is modeled by the AC process LightSrv(s). The internal state of the component is represented by the variable s that records the current operation mode of the connected sensors. The processes CC1(s) and CC2(s) are the component configurations.

To keep the model in reasonable size for presentation purposes, we model the guards of the configurations as always evaluating to true here. The internal state of the light system is made observable using the channel *state*. The

```
channel state : occState . lampState

-- intermediate model of stateless
-- occupancy detection service
OccSrv = camImage?x -> (
    x & detect!camera -> OccSrv
      [] not x & detect!motion -> OccSrv
    )

-- intermediate model of stateless
-- lamp control service
LampSrv = dimValue?x -> (
    x & lamp.dim -> LampSrv
  [] not x & lamp.switch -> LampSrv
    )

-- intermediate model of light control
-- service
-- s = (cam X dim)
LightS = LightSrv((camera, dim))
LightSrv(s) =
  let
      f((_, y), x) = (x, y)
      g((x, _), y) = (x, y)
      guard(s) = true
      status((x,y)) = state.x.y
      CC1(s) = guard(s) & status(s)
               -> LightSrv(s)
      CC2(s) = guard(s) & status(s)
               -> LightSrv(s)
  within (
       detect?x -> CC1(f(s,x)) [] CC2(f(s,x))
    [] lamp?x   -> CC1(g(s,x)) [] CC2(g(s,x))
  )

Intermediate = (OccSrv ||| LampSrv)
              [|{|detect,lamp|}|] LightS
```

Figure 8.4: Intermediate Model of the Light Control System

introduction of observation events is a common method when analyzing CSP models. It can of course be shown, using hiding and refinement, that these events do not change the behavior of the original model. The adaption behavior of the component depends on the adaption eventsdetect?x and lamp?x. After performing one of the events from $\{|\, detect, lamp \,|\}$, the *LightSrv* process evolves to the respective *CC* guarded by *true*. This trivial guard can be removed, of course, but is included for completeness of the *AC* model. The behavior of the *CC*s is identical: both configurations simply communicate the current state of the light control service. Before the LightSrv(s) can engage in further adaptions, the internal state s is updated using the state transformation functions f and g.

```
-- stability
assert Intermediate \AdaptionEvents
       :[livelock free]

-- deadlock freedom
assert Intermediate :[deadlock free]

-- Intermediate refines SPEC
assert SPEC [T= Intermediate\{|state|}
assert SPEC [F= Intermediate\{|state|}

-- SPEC does not refine Intermediate
assert not Intermediate\{|state|} [F= SPEC
```

Figure 8.5: Properties of the Light Control System

The entire system model of the light service and the connected sensors is modeled by the process Intermediate. The two sensors operate in parallel and are not synchronized since they share no adaptive communication channels. The sensors are synchronized with the light service by synchronization on the adaption events *detect* and *lamp*.

Figure 8.5 shows the properties that have been verified for the model using FDR. The first assertion shows that the adaptive system model is stable. Deadlock freedom is checked using the second assertion. The third and the forth assertion express that the adaptive system model Intermediate is both a trace and a failure-divergence refinement of the adaption specification. The last assertion formalizes that the specification does not refine the adaption model.

In Figure 8.6, we extend the previous model of the light control service by introducing dependencies between the state of the camera and the state of the dimmer such that the dimmer is only used if the camera is turned off and the dimmer is available. This behavior is modeled by modifying the behavior of the component configurations and the guards. The refinement demonstrates that using our approach, the adaption behavior of a system can be successively developed and verified.

The following LTL formula captures the property that the dimmer is only used if the camera is turned off and the dimmer is available:

G **not** [state**.**camera**.**dim]

It can be verified using the LTL model checking capabilities of ProB. The assertions presented in Figure 8.7 establish that hiding the observation events ($\{|\, state, dimUnused \,|\}$) again yields a process refining our specification SPEC. In fact,

$$RAM \setminus \{|\, state, dimUnused \,|\} = Intermediate \setminus \{|\, state \,|\}\,,$$

```
-- intermediate model of light control service
-- s = (cam X dim)
channel dimUnused
RAM = let
 LightSrv(s) =
   let
      f((_, y), x) = (x, y)
      g((x, _), y) = (x, y)
      guard((camera,dim)) = false
      guard((x,y)) = true
      status((x,y)) = state.x.y
      CC1(s) =   guard(s) & status(s)
                 -> LightSrv(s)
              [] not guard(s) & dimUnused
                 -> LightSrv(s)
      CC2(s) = CC1(s)
      within (
         detect?x -> CC1(f(s,x)) [] CC2(f(s,x))
      [] lamp?x   -> CC1(g(s,x)) [] CC2(g(s,x))
      )
within (
OccSrv ||| LampSrv) [|{|detect,lamp|}|] LightSrv((camera, dim))
```

Figure 8.6: Refined Adaption Model of the Light Control System

```
assert SPEC [F= RAM\{|state,dimUnused|}
assert RAM\{|state,dimUnused|}
   :[deterministic]
assert Intermediate\{|state|}
   :[deterministic]
assert Intermediate\{|state|}
   [FD= RAM\{|state,dimUnused|}
assert RAM\{|state,dimUnused|}
   [FD= Intermediate\{|state|}
```

Figure 8.7: Properties of the Refined Functional Model of the Light Control System

since both processes are free of non-determinism and cannot be refined further. We proved these statements using FDR.

8.6 Summary

In this chapter, we have presented a modeling approach for distributed adaptive systems. It is based on the idea that a distributed adaptive system can be understood as a reactive system. Based on receiving an event, which signals that adaption is necessary, the system decides to change its behavior and executes certain adaption processes. We have illustrated the modeling approach using a small example.

The modeling approach presented in this section demonstrates that adaptive systems can be elegantly modeled and analyzed using CSP. In combination with our approach for the specification and verification of conformance relations between high-level specifications in CSP and their low-level implementations, we can ensure that adaption properties established on the abstract level also hold for the implementation.

In the next chapter, we will use the presented strategy to model and analyze a distributed and adaptive scheduling system. For this system, we also derive a low-level implementation and explain how our conformance relation can be applied in this setting.

9 Case Studies

In this chapter, we evaluate our verification environment using two case studies. We demonstrate how our verification environment supports the gapless specification and verification of component-based distributed systems from abstract specifications down to low-level implementations. Moreover, we show that the mechanical support offered by our verification environment is of great help even for relatively small systems.

First, we use our verification strategy to verify the implementation correctness of a distributed adaptive scheduling system. On the abstract level, we specify the system using Timed CSP specifications. These focus on the adaption correctness of the system. Furthermore, we have proved functional correctness properties about the system. We then verify the conformance of a prototypical low-level implementation for the scheduling component of the system. The implementation of the scheduler is given in the timed low-level language defined in Chapter 6.2.

As a second case study, we extend the basic low-level language defined in Chapter 5 to a subset of the LLVM IR. We formalize a memory model, which supports reasoning about pointer operations. Using the low-level implementation of a function computing the factorial of a given input, we demonstrate that mechanized total correctness proofs are feasible using our verification environment.

9.1 A Distributed Scheduling System

In this section, we use our verification environment to specify and and analyze the behavior of a distributed scheduling system. We start by explaining the specifications of individual system components and establish properties about the adaption behavior of the composed system. Then, we establish conformance between the specification of the scheduling component and a prototypical implementation given in the timed low-level language from Chapter 6.2.

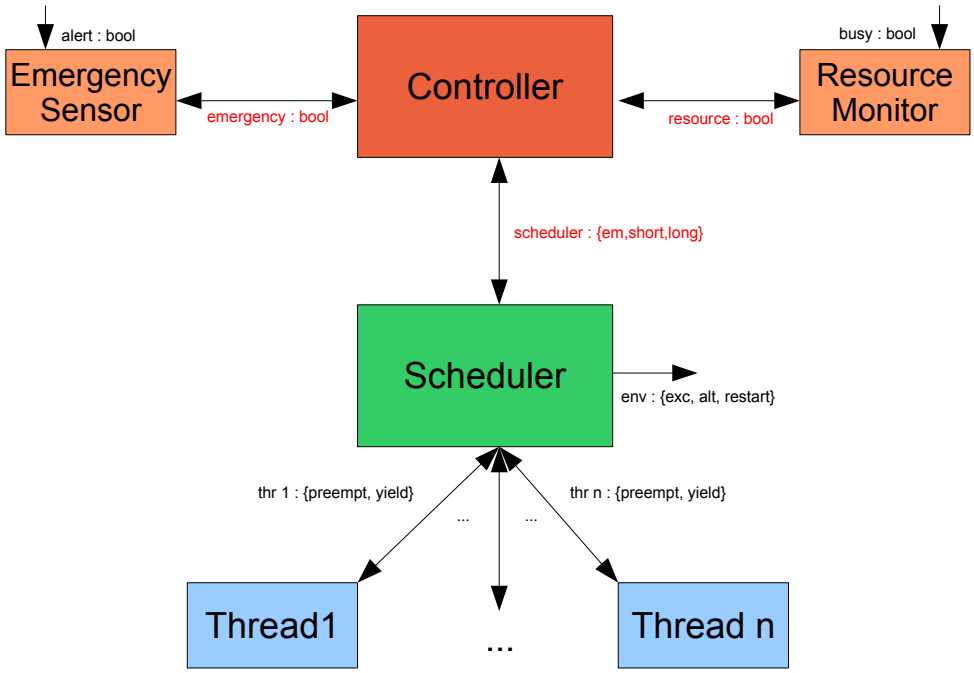

Figure 9.1: Structural Overview of the Distributed Scheduling System

9.1.1 Specifications of the System

A structural overview of the system is given in Figure 9.1. The main components of the system are a controller and a scheduler. The distributed character of the system is given by the interaction of these two components because it determines the scheduling strategy and the adaptive behavior of the system. The controller is connected with two sensors. These monitor the environment of the system. The *Emergency Sensor* detects emergency situations and the *Resource Monitor* monitors the communication and computation facilities that are available for the system to operate and for the threads to perform their computations. Depending on the signals that the two controllers receive from the connected sensors, the controller decides in which configuration the scheduler is supposed to work and communicates the required mode of operation to the scheduler. This reflects the adaption behavior of the distributed scheduling system. The reaction of the system to changes in the environment can be categorized into three configurations. These are reflected by the configuration processes that the scheduler performs. If an emergency situation is detected, the controller notifies the scheduler that it needs to perform its emergency process. If the system detects that the available resources have changed, the controller signals to the scheduler that it needs to change the way of scheduling the threads it controls: the scheduler either gives the threads a short interval of 14 time units to perform their tasks or a longer interval of 28 time units. The scheduler component implements a basic round based fixed priority scheduling mechanism to control the n threads it is connected to.

On the specification level, we follow our strategy for the specification and verification of adaptive systems presented in the last chapter. We first focus

on the untimed adaption behavior of the system by giving adaptive specifications and an adaptive system model for the controller component. For the scheduling component, we also provide an implementation model focussing on the real-time behavior. This implementation model is used to derive a prototypical low-level implementation. For the model and the implementation we show conformance between the two representations using our approach for the verification of conformance relations from Chapter 7.

Adaptive and Non-adaptive Specifications

In this section, we explain the behavior of the scheduling system in more detail. We start by explaining the adaption behavior of the system, i.e., we focus on how the system changes its configurations due to changes in the environment and give a process algebraic specification that expresses this behavior. Then, we explain the non-adaptive behavior of the system. We focus on the behavior of the scheduler.

Adaptive Specification Adaptive specifications AS as introduced in Chapter 8 express properties about the expected adaption behavior of a system. The specification shown in Figure 9.2 is an example of such a property. The specification expresses that the scheduling mode the scheduler operates in can be changed only if no emergency situation has occurred (symbolized by the event *emergency.true*. If no emergency situation was signaled, the scheduler can be switched between two operation modes:

- The short scheduling mode where threads have a time slot of 14 time units to fulfill their task before they get preempted by the scheduler.
- The alternative scheduling mode where threads have a time slot of 28 time units to get their work done before they get preempted.

If the system is required to switch to the first configuration, this is signaled by communicating the adaption event *sched.short*, while adaption to the second configuration is signaled using the event *sched.long*. If adaption of the system is necessary due to an emergency situation (*emergency.true*), the scheduler can only operate in its emergency mode. The necessity to adapt its behavior to this mode is signaled using the adaption event *sched.em*. The first part (*ASpecRes*) of the specification expresses that in reaction to changes in the resources of the system, the scheduling component allows its threads more time to fulfill their task if the resources are busy (indicated by the resource monitor through the adaption event *resource.false*). If the resources are in a good state (indicated by the resource monitor through the adaption event *resource.true*), the threads are assumed to be able to fulfill their tasks in a shorter time slot. These changes of the schedulers configuration according to the state of the environment discovered by the *Resource Sensor* can arbitrarily alternate. Furthermore, notifications of the *Emergency Sensor* signaling that no emergency situation is given can be communicated in between. However, if the *Emergency Sensor* detects that an emergency situation has arisen, the emergency

$$
\begin{aligned}
ASpecRes = (&resource.true \rightarrow sched.short \rightarrow ASpecRes) \\
&\square (resource.false \rightarrow sched.long \rightarrow ASpecRes) \\
&\square (emergency.false \rightarrow ASpecRes) \\
&\square (emergency.true \rightarrow sched.em \rightarrow ASpecEm)
\end{aligned}
$$

$$
\begin{aligned}
ASpecEm = (&ermergency.true \rightarrow ASpecEm) \\
&\square (emergency.false \rightarrow ASpecRes)
\end{aligned}
$$

Figure 9.2: Part of the Adaptive Specification of the Scheduling System

routine of the scheduler needs to be performed (indicated by *sched.em*). After this, only emergency events are allowed until the event *emergency.false* is communicated (*ASpecEm*) .

Non-adaptive Specification In Chapter 8, we have distinguished adaptive from functional specifications. The term functional specification comprises properties that are not related to the adaption behavior of the system. However, the term functional might be slightly misleading in the setting of our case study because timing properties of the scheduler are not related to the adaption behavior, but in general are called non-functional properties. Therefore, we use the term non-adaptive properties here. For the implementation model of the scheduler given in Figure 9.4, we have shown that the scheduler respects the priorities of threads: after a thread with a certain priority is started, a thread with a higher priority can only be started in the next scheduling round. The start of a new scheduling round is indicated by the event *sched.restart*. Another property we have verified is that the scheduler implements correct time slices, i.e., that in short mode the earliest time at which preemption of threads can happen is 24 time units after the respective thread is started (the 14 time units of the specified timeout plus 8 time units for the execution of instructions leading to the output instruction responsible for the preemption), while in long mode the time slice is 38 time units long. As a liveness property, we have shown that the scheduler is able to react to adaption events at least every 28 time units when operating in short mode. We have verified these properties for instances of our distributed scheduling system using the model checking capabilities of FDR3 by taking lists of threads as fixed. Verification for arbitrary numbers of threads can be done using the verification framework for parameterized systems presented in [Göt12].

Adaptive System Model

Following our strategy for the specification of adaptive systems, we first focus on realization of the adaption behavior in terms of configurations by defining

$$
\begin{array}{l}
EmergencySensor = alert?x \rightarrow \\
\qquad (x\&emergency!true \rightarrow EmergencySensor \\
\qquad \Box (not\ x)\&emergency!false \rightarrow EmergencySensor) \\
ResourceMonitor = busy?x \rightarrow \\
\qquad (x\&resource.good \rightarrow ResourceMonitor \\
\qquad \Box (not\ x)\&resource.bad \rightarrow ResourceMonitor)
\end{array}
$$

$$
\begin{array}{l}
CHelp((s1, s2, s3)) = emergency.true \rightarrow CHelp((s1, s2, d3)) \\
\qquad\qquad \Box\ emergency.false \rightarrow Contr((false, s2, false)) \\
Contr(s) = \textbf{let} \\
\quad status((x, y, z)) = state.x.y.z \\
\quad cc1guard((true, \_, \_)) = false \\
\quad cc1guard((false, true, false)) = true \\
\quad cc1guard((false, true, true)) = false \\
\quad cc1guard((false, false, \_)) = false \\
\quad cc2guard((true, \_)) = true \\
\quad cc2guard((false, \_)) = false \\
\quad cc3guard((true, \_, \_)) = false \\
\quad cc3guard((false, true, false)) = false \\
\quad cc3guard((false, false, true)) = true \\
\quad cc3guard((false, false, true)) = false \\
\quad cc4guard((false, \_, true)) = true \\
\quad cc4guard((false, \_, false)) = false \\
\quad cc4guard((true, \_, \_)) = false \\
\quad fem((s1, s2, s3), x) = (x, s2, true)\ \ fbl((s1, s2, s3), x) = (s1, x, false) \\
\quad CC1(s) = cc1guard(s)\&status(s) \rightarrow sched.short \rightarrow Contr(s) \\
\quad CC2((s1, s2, s3)) = cc2guard((s1, s2, s3)))\&status((s1, s2, s3)) \\
\qquad\qquad\qquad \rightarrow sched.em \rightarrow CHelp((s1, s2, s3))) \\
\quad CC3(s) = cc3guard(s)\&status(s) \rightarrow sched.long \rightarrow Contr(s) \\
\quad CC4(s) = cc4guard(s)\&status(s) \rightarrow Contr(s) \\
\textbf{within} \\
(emergency?x \rightarrow \\
(CC1(fem(s, x)) \Box CC2(fem(s, x)) \Box CC3(fem(s, x)) \Box CC4(fem(s, x)))) \\
\Box \\
(resource?x \rightarrow \\
(CC1(fbl(s, x)) \Box CC2(fbl(s, x)) \Box CC3(fbl(s, x)) \Box CC4(fbl(s, x))))
\end{array}
$$

$$
\begin{array}{l}
Scheduler(s) = \textbf{let}\quad cc1guard(s) = (s == em) \\
\quad cc2guard(s) = (s == long)\quad\ cc3guard(s) = (s == short) \\
\quad CC1(s) = cc1guard(s)\&env.exeception \rightarrow Scheduler(s) \\
\quad CC2(s) = cc2guard(s)\&env.busy \rightarrow Scheduler(s) \\
\quad CC3(s) = cc3guard(s)\&env.nonbusy \rightarrow Scheduler(s) \\
\textbf{within}\ (sched?x \rightarrow (CC1(s) \Box CC2(s) \Box CC3(s)))
\end{array}
$$

Figure 9.3: Adaptive System Model of the Scheduling System

an adaptive system model *ASM*. This model is focused on the implementation of the adaption processes within the components of the system. Figure 9.3 shows the processes of the *ASM*.

The first two processes *EmergencySensor* and *ResourceMonitor* model the sensors that monitor the environment of the scheduling system. Both work in one configuration and have one input channel (*alert* and *busy*) that is used to communicate with the environment. Depending on the changes in the environment, which are reflected by the respective input events, the sensors communicate to the other components that the systems needs to adapt its behavior via their adaption channels (*emergency* and *resource*). The third process specifies the behavior of the controller component. Based on the adaption messages that the controller receives from the connected sensors, it decides in which configuration the system works. The controller is implemented using an internal state s, which is formalized as a tuple consisting of three parts. The first component reflects the emergency state, while the second reflects the resource usage. The third one is used to store if the last received event was an emergency event. Based on the messages the controller receives through its adaptive channels and its current state, the controller then decides in which mode the scheduler works. It does so by communicating the respective adaptive events from the control configurations *(CC1, CC2, CC3,CC4)* that are available. The adaptive behavior of the scheduler *Sched* is specified using three control configurations. The process specifies that the scheduler changes its configuration based on the operation mode (*em, long, short*) it receives through the adaptive channel *sched*. The scheduler then communicates to the environment in which mode it is working through the channel *env*.

We established that the combined *ASM* is correct with respect to the property *ASpecRes* defined in Figure 9.2 using the model checker FDR3. More precisely, it refines the specification in the failures-model. Furthermore, the combined *ASM* is deadlock-free. As the next step in our implementation strategy, we add non-adaptive behavior to the description of our system by defining an implementation model for the scheduler component.

Implementation Model

We focus on the specification of the implementation model for the scheduler. Figure 9.4 shows the Timed CSP specifications of the scheduler component. This specification is parameterized over two lists of threads. The list of threads *tlist* is used by the scheduler to keep track of the threads that still need to be considered within the current scheduling round, while *ilist* is the initial list of threads at the start of every round. Furthermore, the model of the scheduler uses the Boolean flag *sFlag* to keep track of its current scheduling mode (short or long). The process can initially either perform its emergency procedure, or switch to the scheduling mode it is currently not operating in. If *sFlag* is set, the process offers the event *sched.long* to its environment, if *sFlag* is not set then the scheduler offers to synchronize on *sched.short*. If the scheduler engages in one of these events it signals the change of operation mode to the environment by communicating *env.busy* or *env.nonbusy* and sets *sFlag* to the new value. Note that by making this implementation decision, the scheduler does not initially offer both operation modes anymore, which might lead to deadlock situations when it is composed with the rest of the

$$
\begin{array}{l}
Sch(tlist, ilist, sFlag) = \\
WAIT(1); \\
\quad ((sched.em \rightarrow WAIT(4);\ env.exception \rightarrow WAIT(3);\ Sch(tlist, ilist, sFlag) \\
\quad \Box \\
\quad (\textbf{if}\ sFlag\ \textbf{then}\ (sched.long \rightarrow WAIT(6);\ env.busy \rightarrow WAIT(3); \\
\qquad\qquad Sch(tlist, ilist, False))) \\
\qquad \textbf{else}\ (sched.short \rightarrow WAIT(6);\ env.nonbusy \rightarrow WAIT(3); \\
\qquad\qquad Sch(tlist, ilist, True))) \\
\quad \Box \\
\quad (\textbf{if}\ (tlist = []\ )\ \textbf{then}(\ (sched.restart \rightarrow WAIT(8); \\
\qquad\qquad env.restart \rightarrow WAIT(3);\ Sch(ilist, ilist, sFlag))) \\
\quad \textbf{else} \\
\quad ((thr(\textbf{hd}\ tlist)).start \rightarrow WAIT(8); \\
\quad\ (((thr\ (\textbf{hd}\ tlist)).yield \rightarrow WAIT(3);\ Sch(tl\ tlist, ilist, sFlag)) \\
\quad\ (\textbf{if}\ sFlag\ \textbf{then}\ 14\ \textbf{else}\ 28) \\
\qquad\qquad \triangleright \\
\quad\ ((WAIT(2);\ (thr\ (\textbf{hd}\ tlist)).preempt \rightarrow WAIT(3); \\
\qquad\quad Sch(insort\ prio\ (\textbf{hd}\ tlist)\ (\textbf{tl}\ tlist), ilist, sFlag)))))))
\end{array}
$$

Figure 9.4: Implementation Model of the Scheduler

system. Therefore, a refined model of the controller is required. In this refined model, the controller only propagates changes in the operation mode, if the new mode is different from the last one. However, the refined controller model is even bigger than the one presented above, so we omit the formal model here for brevity. The third event that is initially possible for the scheduler to synchronize on depends on the list of threads *tlist* that is still considered in the current scheduling round. If the list is empty, the scheduler signals that the current round is finished and restarts with the initial list *ilist*. If there are still threads remaining to be scheduled, the scheduler performs its natural task of governing threads. It picks the first thread in the list of threads ordered by priority and gives it the opportunity to start. Then, depending on its current mode of operation, the scheduler grants either 14 or 28 time units to the running thread to perform its task. If the respective thread has finished its task for the current round, it can yield the right for execution. If it does so, the scheduler drops the thread from its list *tlist* of threads considered for the current scheduling round. If it does not the thread only has the possibility to synchronize on the preemption event. Afterwards, the scheduler inserts the id of the respective thread into its list *tlist* at the first position after all threads with the same priority. This is realized by the function *insort*. Subsequently, the scheduler offers the initial choices again.

Based on this implementation model, we develop a low-level implementation for the scheduler. Therefore, the implementation model given here is also deployed for the conformance proof and thereby ensures that the adaption and the non-adaption properties established using the implementation model hold for the low-level implementation.

```
1   input changeModeOrStartFirstThread setFlags alwaysZero 0
2   brt emergencyFlag 9 3
3   brt changeModeFlag 11 4
4   brt restartFlag 13 5
5   input threadYield dropThread timeoutMode 7
6   br 1
7   input threadPreempt insertSorted alwaysZero 0
8   br 1
9   out emergencyScheduling alwaysZero 0
10  br 1
11  out signalModeChange alwaysZero 0
12  br 1
13  out signalRestart alwaysZero 0
14  br 1
```

Figure 9.5: Basic Low-Level Implementation of the Scheduler

9.1.2 Low-Level Implementation

In this section, we derive a prototypical low-level implementation of the scheduling system presented in the last section. We provide an implementation using our basic low-level language. We concentrate on the low-level branching structure by abstracting from memory related operations, for example we use HOL lists in the actual scheduler implementation to keep track of the threads in the system. In a next development step, such an implementation could serve as a prototypical implementation to derive an implementation in a more concrete low-level language.

Prototypical Low-Level Implementation

We focus on the basic low-level implementation of the scheduler component. The goal is to derive a low-level implementation from the abstract specification as given by the implementation model presented in the last section (process $Sch(\_,\_,\_)$ in Figure 9.4). As an implementation language, we use our timed low-level language from Chapter 6. We first define the notion of state. It consists of a function that maps from `nat` to `nat` modeling the available registers and their contents. Furthermore, the state consists of two HOL lists, which model arrays that hold the current order of active threads. The scheduler performs its round based scheduling using the list *tlist* and uses the second list *ilist* to restore the initially ordered list of threads when it performs a new scheduling round. We use HOL lists here to keep our example in reasonable size for presentation purposes. The lists could be exchanged by other memory related concepts in later development steps.

type-synonym $store = (nat \Rightarrow nat) \times (nat\ list) \times (nat\ list) \times bool$

Figure 9.5 shows the basic low-level implementation of the scheduler component. This implementation resembles the implementation model as shown in

Figure 9.4 in the sense that the control flow of the low-level code reflects the order of possible events in the Timed CSP specification. Here, we use HOL functions to model manipulations of the system store, for example when manipulating the lists of the low-level programs store. In the first line, the implementation uses an `input` instruction to realize its interaction with the rest of the system. Based on the current state the function *changeModeOrStartFirstThread* is used to decide if the low-level implementation can switch its mode of operation or if it schedules a thread. Execution of emergency procedures is always possible. After the `input` instruction is performed, the function *setFlags* is used to store which event was performed. The `input` instruction as specified here has a quite abstract behavior, but it can be considered as corresponding to a system call where the system handles communication with other entities and the result is stored in some register. As no timeout is given for the instruction, the timeout value is set to 0 using the function *alwaysZero*. Based on the information about the communication store given by the function *setFlags*, in line 2 to 4 the low-level implementation of the scheduler branches to the label where the chosen functionality is realized. Branching is governed by evaluating the respective Boolean functions. If either a restart or a change of operation mode or the emergency procedure is requested, the control flow continues at the respective label and performs an `output` instruction to signal this to the environment and afterwards execution starts again at label 1. If the implementation is requested to do scheduling, execution proceeds at label 5, where an `input` instruction is used to offer some thread the chance to yield control for either 14 or 28 time units. The respective timeout interval is determined using the function *timeoutMode* based on the current mode of operation stored in the registers. The effect of the `input` instruction on the state, if the specified timeout does not elapse, is that the thread that currently has the control is dropped from the list of active threads. If the timeout elapses, execution proceeds at label 7. The implementation offers preemption and the effect on the state is that the currently running thread is inserted into the list of threads still considered in this scheduling round.

We have formally verified that the prototypical implementation correctly implements the *IM* given in the previous section. We specify a bisimulation relation as explained in Chapter 7 and have proven its correctness using our verification strategy. We have used our specification and verification strategy for the abstraction from the internal behavior of the low-level implementation that is realized by the instruction in lines 2 to 4 in the implementation. For example, we have proven the following total correctness specification using our proof calculus:

$$\begin{aligned}\vdash_t \langle \lambda\ aux\ (s,str).\ & PId\ s{=}1 \wedge PC\ s{=}2 \wedge fst\ (R\ s)\ 1{=}1 \wedge fst\ (R\ s)\ 2{=}0 \wedge \\ & fst\ (R\ s)\ 3{=}0 \wedge fst\ (R\ s)\ 4{=}0 \wedge fst\ (snd\ (R\ s)) \neq [] \wedge \\ & fst\ (snd\ (R\ s)) = TLIST\ aux \wedge fst\ (snd\ (snd\ (R\ s))) = ILIST\ aux \wedge \\ & tnow\ s = TNOW\ aux \wedge str = TRX\ aux \rangle\end{aligned}$$

$$\begin{aligned}&(_{1|}(((_{2::}brt\ emergencyFlag\ 12\ 4)_{\oplus} \\ &\quad (_{3::}brt\ changeModeFlag\ 11\ 4))_{\oplus} \\ &\quad (_{4::}brt\ restartFlag\ 13\ 5)))\end{aligned}$$

$$\langle \lambda\ aux\ (s,str).PId\ s{=}1 \wedge PC\ s{=}5 \wedge fst\ (R\ s)\ 1{=}1 \wedge fst\ (R\ s)\ 2{=}0 \wedge fst\ (R\ s)\ 3{=}0 \wedge fst\ (R\ s)\ 4{=}0 \wedge fst\ (snd\ (R\ s)){\neq}[] \wedge fst\ (snd\ (R\ s)){=}TLIST\ aux \wedge fst\ (snd\ (snd\ (R\ s))){=}ILIST\ aux \wedge tnow\ s{=}TNOW\ aux{+}6 \wedge str{=}TRX\ aux\rangle$$

It covers the situation where a thread was started because the respective event was chosen from the set of events offered by the input instruction in line 1. The function *setFlags* sets the register 1 to 1, so that the control flow branches to line 5 where the started thread is given the opportunity to yield control. The total correctness specification ensures that exactly 6 time units pass by (because all `brt` instructions are executed) and that no communications take place. Note that we use the auxiliary state here to ensure that the lists the scheduler works on and also the communication trace is not changed by the instructions. Furthermore, we have defined a variable *TNOW* in the auxiliary state that we use to ensure that execution of the instructions takes 6 time units. The variant for the sequential compositions can be defined using the program counter since there are not backward jumps in this section of code. Our abstraction theorems then enable us to specify the corresponding situation during the bisimulation proof. They enable us to treat the behavior of the code from line 2 to 5 as if 6 time units pass by. This especially reduces the amount of proof steps necessary in the bisimulation proof. Instead of passing through 2 normal states and all the states between these normal states on the level of the small-step semantics, the proof can go directly from the normal *nstate* where the program pointer points to the label 2 to the *nstate* where it points to the label 5.

The following lemma shows implementation conformance of the scheduler implementation given in Figure 9.5 with respect to its implementation model as given in Figure 9.4. The lemma states that for any list of threads that is ordered according to the threads' priorities, the implementation model and its low-level implementation are semantically equivalent if started with the same initial lists:

lemma *scheduler-conformance* :

$$k > 0 \Longrightarrow let\ tlist = sort\text{-}key\ prio\ [[1..k]]\ in$$
$$(\forall\ s.\ (\lambda\ s.\ PId\ s = 1 \wedge PC\ s = 1 \wedge fst\ (snd\ (R\ s)) = tlist \wedge fst\ (snd\ (snd\ (R\ s))) = tlist\ )\ s \longrightarrow (tcsp\ (asg\ (Sch(tlist,tlist,False))),\ (ll\ (nstate\ s))) \in weak\text{-}timed\text{-}bisimilar)$$

The proof of conformance took us about 35 hours and the Isabelle code to realize it takes about 8000 lines of proof code. This high number demonstrates that without mechanical support such proofs are hardly possible. The amount of proof code is mainly due to the fact that the prototypical implementation of the scheduler contains almost as many communication instructions as internal instructions. For the internal instructions, our specification and verification strategy reduces the number of necessary proof steps dependent on the number of consecutive internal instructions. Once a proven Hoare-triple is available, it takes just 2 applications of our abstraction theorems for each direction of the bisimulation proof for any number of internal instructions. When working directly on the level of the small-step semantics this takes 4 proof steps per

instruction. Recall that every instruction basically corresponds to 3 Timed CSP processes and 4 state transitions. They model the passage of time before the effect of an instruction, the effect of the instruction and the passage of time after the instruction. Our approach to abstract from sequences of internal instructions significantly reduces the amount of proof steps for sections consisting of internal instructions because the required total correctness proofs are realized using the proof calculus. The calculus 'operates' on the level of the big-step semantics, where the 3 processes used on the small-step level are subsumed by just 1 transition. However, for the parts of the bisimulation relation relating to communication instructions our environment is currently in a state of a proof of concept implementation. The reduction of necessary proof steps using our abstraction theorems motivates further abstraction theorems targeting communication instructions. We expect to be able to further lessen the burden of manual proofs and reduce the amount of proof code using such theorems.

In this section, we have evaluated our entire verification environment using a distributed scheduling system. Given an implementation in the real-time low-level language, we have mechanically established conformance between the low-level implementation and an implementation model given in Timed CSP. The proof of bisimulation became quite complex due to the low-level semantics of the real-time low-level language. Using our abstraction theorems, we were able to significantly reduce the amount of work required to account for internal transitions in the bisimulation proof.

In the next part of this section on case studies, we want to evaluate our verification environment in the setting of total correctness proofs for conventional, i.e., untimed low-level code. To provide a realistic implementation language, we instantiate our basic low-level language and the related proof calculus with a memory model. We mainly concentrate on the feasibility of mechanized total correctness proofs for more complex low-level languages in this case study.

9.2 Instantiation to a Subset of the LLVM IR

In this section, we use the the semantics and the proof calculi presented in Chapter 5 as the basis for the formalization of a subset of the LLVM IR. The main goal is to evaluate the practical feasibility of termination proofs for code given in the intermediate representation. The targeted subset contains instructions for integer and real arithmetics and the LLVM IR's instructions for memory manipulation. We give a brief overview of the memory model and the semantics for the chosen subset and then prove the total correctness for a small low-level program given in the chosen subset.

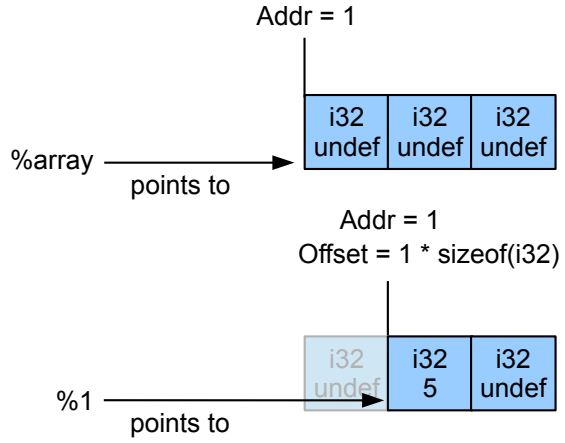

Figure 9.6: Allocation of Memory

9.2.1 Memory Model and Instruction Set

To associate variable names in programs with values in memory, we use a memory model that follows the one used in [BL05] to formalize the semantics of an intermediate language for a moderately optimizing C compiler. The advantage of the model is that reasoning about memory structures partly avoids the need for assertions ensuring that certain regions within the memory layout do not overlap because memory is allocated in terms of blocks. However, this mechanism only separates memory on the granularity level of blocks and is thus not as powerful as dedicated logics such as separation logic [ORY01].

We use the following record m to model the memory that the considered LLVM IR subset operate on:

record $m =$
$Addr :: addr$
$B :: addr \Rightarrow nat$
$F :: addr \Rightarrow bool$
$C :: (addr \times nat) \Rightarrow content$
$T :: typearg \Rightarrow type$

The first part of the record m is the current memory address $Addr$ at which new memory space can be allocated. If a chunk of memory is allocated, this variable is increased so that it points to the location after the allocated memory space where further space can be allocated. The function B yields the block size for a given memory address at which a chunk of memory was allocated. If a chunk of memory is allocated at an address, the function F is used to indicate this. For a given address, the function C yields the content. Note that memory is addressed using an offset from the allocation addresses. The function T is used to keep track of typing information for pointer variables. Figure 9.6 depicts the way memory and accesses to it are modeled. There, an array consisting of three integer variables is allocated at the location $Addr = 1$. The register $\%array$ of type integer pointer initially points to this address. The content of the second element of the array can be derived using the content

function C. It takes as input the address of the array and the offset that points to the second element of the array. The offset is calculated by multiplication of the number of the respective element with the size of the respective datatype (yielded by the function *sizeof*).

The program state consists of a function R modeling SSA registers, functions S and H that associate variable names with addresses on the parts of the memory that model the stack and the heap, the overall memory structure M as described above and the program counter PC:

record *state* =
R :: $reg \Rightarrow regval$
S :: $var \Rightarrow (addr \times nat)$
H :: $var \Rightarrow (addr \times nat)$
M :: m
PC :: $label$

We formalize the following LLVM IR instructions:

datatype *instructions* =
iassign reg intset |
br label |
brt reg label label |
add reg type reg reg |
add-const reg type reg int |
mul reg type reg reg |
icmp-slt reg type reg reg |
alloca ident type |
load reg type ident |
store-const type content type ident |
store type reg type ident |
gep reg type reg gepType

The first instruction `iassign` is not part of the LLVM IR specification. However, we use it here to model non-deterministic input behavior, which on the level of low-level languages is realized by calls to functions provided by the operating system. Here, it assigns the values from a set of integer values to a specified register, i.e., it yields a state for every value in the respective set. The syntax of the two branching instructions is similar to the syntax of our basic low-level language. The instruction `add` adds the contents of two registers that have the same *type* and stores the result to another register, while `add_const` adds a constant to the value in a given register and stores the result in a further register. The instruction `mul` is similar to `add`, but multiplies the contents of two given registers. The instruction `icmp_slt` compares the values of two registers of type *type* and stores a Boolean result in a further register (*True* if the value in the first register is smaller than the value in the second register and *False* otherwise). The remaining instructions are concerned with memory operations. The instruction `alloca` allocates a block of memory on the current stack frame, while `load` copies the value stored in heap or stack space to a register. The instruction `store_const` is used to store some value to a memory location identified by a pointer *ident*, while the instruction `store`

can be used to store a value residing in a register `reg` to a location in memory that is indexed by a pointer *ident*. The last instruction `gep` is used for pointer operations. It can be used to reference datatypes within array-like structures on the stack. Given a pointer to an array structure, the type of the element and an offset value, it yields the address of the respective element.

Based on the specified subset of instructions, we define small-step and big-step semantics based on our low-level language from Chapter 5. The main challenge is to cope with the necessary instantiations of the assignment rule in order to read and write to the memory. Once the small-step semantics is defined, the definition of the big-step semantics and the total correctness calculus are straightforward. However, the definition of the termination predicate needs to be adjusted with care. Using our subset of the LLVM IR, we can now realize total correctness proofs about programs that operate on a sophisticated memory layout.

9.2.2 Specification and Verification

As a small example, we verify the total correctness for a small implementation given in the above defined subset of the LLVM IR. Figure 9.7 shows the implementation of the factorial function. It represents a program that first allocates memory space for the local variables n, x and s on the runtime stack. Then, execution of the program proceeds by non-deterministically reading a variable from the set *IS* and storing it into the register 0. From this register, the value is stored into a variable residing on the stack with the name n. Two further variables are used to compute the factorial of n, these are x and s. They are initialized in lines 6 and 7. After this initialization, execution continues by jumping to the label 17. From there, the variables x and n are loaded into the registers 6 and 7 in order to check whether the value of x is smaller than the value of n. The outcome of the comparison is stored in register 8. If x is smaller, execution proceeds at label 9 where the code corresponding to the loop body is entered. If x is not smaller, execution stops at label 21.

For the given code, we prove that it calculates the factorial for any value that is read from some unspecified input source, for example a sensor. This input from an unspecified source is modeled using the command *iassign* in line 4. Furthermore, we want to ensure that execution of the function always terminates. The total correctness specification we want to prove is the following:

$$
\begin{aligned}
&\vdash_t \langle \lambda aux\ s.\ PC\ (last\ s) = 1 \rangle \\
&\quad code \\
&\quad \langle \lambda aux\ s.\ PC\ (last\ s) = 21\ \wedge \\
&\qquad\qquad (cint2nat\ (C\ (M\ (last\ s))((S\ (last\ s))\ ''s'')))\ = \\
&\qquad fac\ (cint2nat\ (C\ (M\ (last\ s))((S\ (last\ s))\ ''x''))) \rangle
\end{aligned}
$$

It states that given a state where the program pointer points to the label 1, execution of the program from Figure 9.7 ends in a state where the program counter points to the label 21. The memory location that the variable s points

```
 1  alloca (idvar ''n'') tint
 2  alloca (idvar ''x'') tint
 3  alloca (idvar ''s'') tint
 4  iassign 0 IS
 5  store tint 0 (tptr tint) (idvar ''n'')
 6  store_const tint 0 (tptr tint) (idvar ''x'')
 7  store_const tint 1 (tptr tint) (idvar ''s'')
 8  br 17
 9  load 1 (tptr tint) (idvar ''x'')
10  add_const 2 tint 1 1
11  store tint 2 (tptr tint) (idvar ''x'')
12  load 3 (tptr tint) (idvar ''s'')
13  load 4 (tptr tint) (idvar ''x'')
14  mul 5 tint 3 4
15  store tint 5 (tptr tint) (idvar ''s'')
16  br 17
17  load 6 (tptr tint) (idvar ''x'')
18  load 7 (tptr tint) (idvar ''n'')
19  icmp_slt 8 tint 6 7
20  brt 8 9 21
```

Figure 9.7: LLVM IR representation of the factorial function

to holds a value that is the factorial of the value in memory that the variable x points to. The identifier *last* appears in the assertion because we have based our semantics on traces of states. For every state transition a state is appended to the trace. The last state of such a state trace is then the current state. Note that we use the total correctness calculus here (indicated by the symbol $\vdash_t$). This requires the specification of a variant. For straight line code, we use the inverse of the program counter. For the part of the low-level code that realizes the loop behavior, we use the following variant:

$$\begin{array}{l} \mathit{loop\text{-}var\text{-}fun}\ s = \\ ((\mathit{cint2nat}\ (C\ (M\ (\mathit{last}\ s))((S\ (\mathit{last}\ s))\ ''n''))) - \\ \ (\mathit{cint2nat}\ (C\ (M\ (\mathit{last}\ s))((S\ (\mathit{last}\ s))\ ''x''))) \end{array}$$

The variant reflects the fact that the loop body (in which x is incremented) is executed until x equals n. The proof of functional correctness corresponds to the standard proof for high-level implementations of the factorial function. The help variables x and s are initialized to 0 and 1, so initially the value of s is the factorial of x. Then in every loop iteration, x is incremented and s is multiplied with x. This already yields the invariant: after every loop iteration, the value of s is the factorial of the value in x and x is smaller than n. When execution of the loop does not continue because the loop condition is falsified, we know that s already holds the calculated factorial of n because x cannot be greater than n and furthermore x and s either hold their initial values or were manipulated by executions of the loop body, which preserve the relation between x and s.

The proof code for the verification of the example function is about 1500 lines long. This includes the specification of the variants and invariants for the total correctness proofs that are quite verbose. Furthermore, intermediate specifications arising from the application of the rule of consequence contribute to the sheer size of the proof code. Regarding the proof obligations that need to be discharged, we greatly benefit from the automatization features of Isabelle for its instantiation to HOL. Based on the experiences with our prototypical verification environment, we are confident that the complexity and amount of proofs can be massively decreased by the implementation of a verification condition generator (vcg) as used for a high-level language in [Sch06]. Such a tactic applies the rules of the proof calculus in an automatized way and therefore reduces the amount of manual work necessary to apply proof rules and specify intermediate assertions. Our instantiation of the basic low-level language to a subset of the LLVM IR is an important starting point for a mechanized and automatized verification environment that enables concrete proofs about practically relevant low-level implementations.

9.3 Summary

In this chapter, we have evaluated our verification environment using two case studies. As the first case study, we have verified the correctness of a distributed and adaptive real-time scheduling system. The main part of the scheduling system is a scheduler that is supposed to start and stop threads. The scheduler component is connected to a controller that is used to keep track of problems with the communication infrastructure and other emergency situations. Based on the information about the infrastructure, the distributed system decides in which configuration it operates. Using our verification strategy for the specification and verification of adaptive systems, we were able to show that the adaptive behavior and the real time behavior of the scheduler and its connected components satisfy the specified requirements. We have not only verified this on the abstract level, but we have used our conformance relation to show that the low-level implementation of the scheduling component also satisfies the high-level requirements. By combining automated model checking methods on the abstract level with interactive theorem proving techniques for mechanized conformance proofs, we were able to transform individual components of the distributed system into their low-level representations in a stepwise fashion. Due to our mechanized proofs of conformance, we are able to claim that the partially transformed system preserves the desired properties as specified on the abstract level.

As the second case study we have verified the total correctness of a program given in a subset of the LLVM IR. We have presented our formalized subset of the IR together with its memory model. Then, we have used our proof calculus for total correctness proofs of low-level code to verify the total correctness of an implementation that calculates the factorial of a non-deterministic input. The example shows that, using our formalization for low-level code, we can realize total correctness proofs about realistically modeled intermediate code

and support them with the mechanisms provided by Isabelle/HOL for the construction of well-founded relations.

In the next chapter, we conclude this thesis. We review the concepts presented in the preceding chapters with respect to the objectives from the introduction and discuss possible directions for future work.

10 Conclusions and Future Work

In this chapter, we conclude this thesis by summarizing and discussing the obtained results. In Section 10.1, we summarize our developed concepts for the mechanical verification of conformance between high-level specifications and low-level code in a real-time setting. Subsequently, we discuss them with respect to the objectives given in Section 1.2. Our mechanical verification environment can serve as a basis for a variety of extensions, which constitute interesting directions for future work. These are discussed in Section 10.3. Finally, in Section 10.4, we summarize which parts of this thesis have already been published in international conference proceedings.

10.1 Conclusion

In this thesis, we have presented a verification environment for conformance proofs in the setting of low-level code and process-algebraic specifications. Our work is motivated by the observation that even though in many software development processes, representations of a system are modeled and analyzed on different levels of abstraction, support for formally funded conformance relations is still limited. This is especially true in the setting of embedded real-time systems.

To address this problem, we have designed an environment that supports high-level representations given as timed process-algebraic specifications and implementations given in terms of low-level code. For the analysis of process-algebraic specifications our framework builds on a formalization of the process-algebra Timed CSP. For the specification and verification on the level of the low-level code, we have constructed a compositional semantics and a dedicated proof calculus that supports total correctness proofs in the presence of possibly unbounded non-determinism. Moreover, we have extended the semantics and the proof calculus to a real-time setting. This extension includes a small-step semantics, which can be interpreted as defining a timed labeled transition system. It is this representation that enables the application of weak timed bisimulation as a conformance relation in our setting. By using labeled transition systems as an intermediate layer for the application of our conformance relation, we obtain a modular verification environment. It can be extended

with languages and formalisms, which can be interpreted as labeled transition systems. Using weak timed bisimulation, we can relate Timed CSP specifications to their low-level implementations. A main source of complexity within such proofs is to cope with operations considered to be internal to a system. We have presented a specification strategy for bisimulation relations building on Hoare triples together with a related proof strategy. Our proof strategy enables the compositional verification of partial and total correctness of low-level implementations. To further ease the verification process, we provide abstraction theorems, which can be used to discharge proof obligations resulting from our abstract specification of bisimulation relations. Furthermore, we presented an approach for the specification and verification of adaptive systems in CSP. This demonstrates that using our verification environment, interesting classes of systems can be tackled. Our environment serves as the basis for the analysis of such systems on the abstract level and on the level of the low-level implementations. Moreover, conformance between these levels can be formally verified. It is well-known that the verification of low-level code is a complex task on its own. The introduction of timing behavior even raises this level of complexity and also complicates conformance proofs between low-level representations and abstract specifications. Our environment copes with these sources of complexity by formalizing the presented concepts using the theorem prover Isabelle/HOL. This not only provides evidence that the presented theoretical concepts are indeed correct, but also yields a mechanized verification environment that can be used for concrete proofs about low-level code on the one side and its correctness with respect to high-level specifications on the other side. Such proofs can be partly automatized using the proof tactics and mechanisms of the theorem prover.

The main advantages of our approach can be summarized as follows:

- We provide a mechanized compositional semantics and dedicated proof calculus for low-level languages that can cope with time and non-determinism and enables total correctness proofs.
- We obtain a modular and mechanized environment for conformance proofs between real-time specifications in Timed CSP and low-level implementations.
- The specification and verification of conformance relations is supported by mechanized abstraction theorems.
- Our approach for the specification and verification of adaptive systems based on configurations can be used to mechanically verify properties about such systems using partly automatized proofs.
- We provide continuous mechanical support for correctness proofs by formalizing our environment using the theorem prover Isabelle/HOL.

In this section, we have summarized the main contributions of our environment. In the next section, we review them with respect to the objectives given in the introduction in Chapter 1.

10.2 Discussion

In this section, we discuss the achievements presented in this thesis with respect to the objectives given in the introduction.

- **Formal specification of system behavior on different levels of abstraction**

  Our environment supports the specification of a systems behavior using the process algebra CSP and its timed extension Timed CSP. These formal languages are tailored for the specification of systems on different levels of abstraction which we have demonstrated in our approach for the specification and verification of adaptive systems and in our case study. Furthermore, we have presented a stack of semantics for timed low-level languages. This stack includes formalized small-step and compositional big-step semantics that enable the concise specification of low-level code behavior. We have demonstrated that this language can serve as the basis for a formalization of a subset of the intermediate representation of the LLVM framework. This intermediate representation can in turn be used as the the target representation for transformation from a variety of high-level languages. Therefore, the formalization of the IR can also be considered as providing a concise representation of the behavior of high-level languages that can be transformed to the intermediate language.

- **Formally justified conformance relation between different levels of abstraction**

  Using the notion of weak timed bisimulation, our verification environment enables us to relate Timed CSP specifications that target different levels of abstraction. It especially supports development processes that start with a high-level representation that abstracts from concrete realizations of behaviors. These representations are shown to be semantically equivalent to more concrete representations by hiding additional behaviors in the more concrete representation in order to establish conformance. In our verification environment, this strategy can further be used to relate abstract specifications to their low-level representations. This is enabled by our timed small-step semantics, which can be interpreted as a timed labeled transition system. Concrete proofs are supported by our abstraction theorems, which are used to infer possible semantic steps from abstract Hoare calculus proofs.

- **Integration of timing behavior on all levels of abstraction**

  Timing behavior is supported by choosing Timed CSP as a formalism for the level of the abstract specification. The process-algebra provides mechanisms to specify timing behavior and therefore supports concise specifications about the timing behavior of communicating processes. On the level of the implementation, we have demonstrated how timing behavior is integrated on all the levels of the semantics stack for our timed low-level language.

- **Support for adaptivity**

  Our verification environment supports the notion of adaptivity as defined in Chapter 8. There, we focus on a notion of adaptive behavior based on configurations that a system operates in. Configurations are changed in response to changes in the environment in which the system is deployed. By supporting the specification of adaptive behavior on the abstract level and establishing that a given low-level implementation conforms to the abstract level, it is ensured that proven properties also hold for the low-level implementation. Therefore, our verification environment supports adaptivity in the sense that it can be specified and analyzed on an abstract level. The analysis results can then be transferred from the abstract level to the level of the low-level implementation, resulting in a provably correct implementation of an adaptive system.

- **Integration with existing and practically relevant formalisms and programming languages**

  For a concrete instantiation of our verification environment, we have chosen the process-algebras CSP and its timed extension Timed CSP. For the low-level representation, we have chosen to formalize a basic low-level representation that can serve as the basis for the formalization of practically relevant low-level languages. We have demonstrated this by formalizing a subset of the intermediate language of the LLVM compiler framework. This concrete low-level language is of great interest both in industry and academics. Through its modular structure and design, it serves as the target for compilation from a variety of high-level languages. By extending the subset, the formalization of the LLVM IR could therefore serve as a verification intermediate representation for a variety of high-level languages.

- **A high degree of mechanization and automatization**

  Our case studies and the additional examples have demonstrated that conformance proofs and also total correctness proofs about relatively small low-level programs are already a challenging task. To ensure correctness and avoid overlooking corner cases we provide a completely mechanized verification environment. Using the proof tactics and mechanisms of the theorem prover, concrete proofs about low-level code and conformance are aided with automated procedures. Furthermore, our modular proof methodology enables a systematic verification process.

- **Modularity and Extendability**

  By defining our conformance relation using timed labeled transition system as a basis, our environment can be extended with other timed formalisms that support the specification of timing behavior and can be interpreted as labeled transition systems. Furthermore, our timed low-level language can serve as the basis for formalizations of further low-level languages. From a technical perspective, our mechanization sup-

ports modularity and extendability by using the concept of locales in Isabelle/HOL for the formalization of the concepts described above.

In this section, we have reviewed the presented verification environment with respect to the objectives from the introduction. We have shown that we meet all the criteria that constitute a mechanized verification environment that is suited for the mechanized verification of embedded real-time systems on multiple levels of abstraction, thereby bridging the semantic gaps present in typical design processes. In the following section, we discuss possible extensions to our verification environment that motivate future work.

10.3 Future Work

In the previous sections, we have summarized the concepts presented in the preceding chapters of this thesis and discussed them with respect to the objectives given in the introduction. In this section, we discuss various extensions that are motivated by the results obtained so far.

Proof Calculi for Low-Level Languages Our extended proof calculus for total correctness proofs of low-level code supports compositional proofs about the behavior of low-level code. Furthermore, total correctness proofs about possibly non-deterministic low-level code are possible. In its current state, the semantics supports the basic instructions of a typical low-level language. As a next step it would be beneficial to formalize concepts for structuring code, like mutually recursive procedures for example. Another interesting extension would be to integrate the proof logic with separation logic, for example. This would enable the compositional verification of low-level code operating on complex memory structures using our calculus.

Proof Calculi for Distributed Low-Level Languages In Chapter 6.5, we have briefly explained how our formalization can serve as a basis for a compositional proof calculus for a distributed low-level language that communicates through channels. In order to facilitate proofs about deadlock-freedom using such a calculus a more fine grained semantics in terms of the treatment of communications is required. The timed traces used in our current formalization are only sufficient to show safety properties, to obtain a compositional proof system for low-level systems that enables proofs about liveness properties as well a more expressive semantics like the failure-divergence semantics [Ros05] would be required. Another extension into a similar direction would be to extend the calculus to a concurrent setting where low-level code synchronizes using shared memory. A compositional approach that could serve as a basis for such an extension would be the rely guarantee method [Jon83].

Conformance Relation In this thesis, we have chosen the notion of weak timed bisimulation as a conformance relation. As shown in the various examples weak timed bisimulation requires that two processes behave equally with respect to timing behavior. In many situations, a more relaxed notion of conformance with respect to timing behavior would add more flexibility when using our framework. A possible candidate for a less strict conformance relation would be the so-called amortized faster than relation [LV06], which defines a bisimulation relation that allows processes to be identified in a more flexible way. When using such a relation, it is possible for two processes to be identified, even if it sometimes takes more time for a process to simulate a given event of the other process.

For CSP-like languages, denotational semantics have always attracted great attention, because they directly support the notion of refinement, .i.e., by using set-theoretic inclusion on the respective semantics representation. Furthermore, these semantics enable the analysis of processes on a more abstract basis when compared to the relatively low level of abstraction inherited by representations based on operational semantics. To obtain more expressive specifications and allow for a more abstract treatment of the semantics of low-level languages, it would therefore be beneficial to model a denotational semantics for the low-level language using for example the timed failure semantics [Sch99]. Formalizing such a semantics and a related proof calculus in a theorem prover would though be a far more complex undertaking compared to the mechanization of an operational semantics. However, as demonstrated in [Sch92] for Timed CSP, the denotational semantics can be characterized using operational representations. It would be interesting to investigate a similar approach for our timed low-level language. Our given semantics can serve as a starting point for such a semantics.

Adaptive Systems The notion of adaptivity as we have introduced and used it in this thesis is based on configurations. An interesting extension, which enables a more flexible approach to the specification and verification of adaptive systems would be the integration of statistical methods. We aim for an extension of our approach, where reactions to changes in the environment are not triggered by discrete events, but may rather be triggered by statistical observations the system makes about its environment. This especially enables more flexible adaption strategies, i.e., how the system decides what reaction is appropriate to which changes in the environment. To support such an extension in our environment, a possible starting point would be to extend the formalization of CSP in our environment to probabilistic CSP [Sei95].

In this section, we have discussed interesting research questions motivated by the results presented in this thesis. In the next section, we given an overview of our publications related to the topics discussed in this thesis.

10.4 Related Publications

Parts of the results presented in this thesis were published in papers on international conferences. In this section, we briefly explain how the papers correspond to the chapters of this thesis. In [BG11, Bar11] a general overview of the envisioned environment for the verification of real-time low-level code with respect to Timed CSP specifications is given. In [BG10], we describe the idea of using bisimulation as a relation to verify the correctness of process-algebraic models extracted from low-level code using an extraction approach [KBG+11] that is not discussed in detail in this thesis. The mechanized verification of low-level code with respect to total correctness specifications is presented in [BJ14]. Our approach to the specification and verification of adaptive system from Chapter 8 is presented in [BK11]. The relation of the work presented in this thesis to the project VATES is described in [GBGK10, BGG10, KBGG09].

List of Figures

List of Definitions

List of Theorems and Lemmas

Theorems

Lemmas

Bibliography

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[AdBO09] Krzysztof R. Apt, Frank de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer Publishing Company, Incorporated, 3rd edition, 2009.

[ADG98] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, 1998.

[ALOR12] Philip Armstrong, Gavin Lowe, Joel Ouaknine, and A. W. Roscoe. Model checking Timed CSP. In *Proceedings HOWARD-60*, 2012.

[AM01] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, September 2001.

[ANY12] Reynald Affeldt, David Nowak, and Kiyoshi Yamada. Certifying assembly with formal security proofs: The case of bbs. volume 77, pages 1058–1074, Amsterdam, The Netherlands, The Netherlands, September 2012. Elsevier North-Holland, Inc.

[ASSV07] Rasmus Adler, Ina Schaefer, Tobias Schuele, and Eric Vecchié. From model-based design to formal verification of adaptive embedded systems. In *Proceedings of the formal engineering methods 9th international conference on Formal methods and software engineering*, ICFEM'07, pages 76–95, Berlin, Heidelberg, 2007. Springer-Verlag.

[Bal06] Clemens Ballarin. Interpretation of Locales in Isabelle: Theories and Proof Contexts. In *MKM*, pages 31–43, 2006.

[Bar78] Iann M. Barron. The transputer. In D. Aspinall, editor, *The Microprocessor and its Application: an Advanced Course: 343*. Cambridge University Press, 1978.

[Ben05] Nick Benton. A typed, compositional logic for a stack-based abstract machine. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, APLAS'05, pages 364–380, Berlin, Heidelberg, 2005. Springer-Verlag.

[BH06] Lennart Beringer and Martin Hofmann. A Bytecode Logic for JML and Types. In *Proceedings of the 4th Asian conference on Programming Languages and Systems*, APLAS'06, pages 389–405, Berlin, Heidelberg, 2006. Springer-Verlag.

[BL05] S. Blazy and X. Leroy. Formal Verification of a Memory Model for *C*-Like Imperative Languages. In Kung-Kiu Lau and Richard Banach, editors, *International Conference on Formal Engineering Methods*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299. Spring, 2005.

[Bru81] Arie Bruin. Goto statements: semantics and deduction systems. *Acta Informatica*, 15(4):385–424, 1981.

[BY04] Johan Bengtsson and Wang Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Preti Nets*, pages 87–124. Springer–Verlag, 2004.

[CH04] Chaochen and Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. SpringerVerlag, 2004.

[Coo78] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 1978.

[dFELN99] D. de Frutos-Escrig, N. López, and M. Núñez. Global Timed Bisimulation: An Introduction. In *FORTE XII / PSTV XIX '99*, pages 401–416. Kluwer, B.V., 1999.

[DHSZ06] Jin Song Dong, Ping Hao, Jun Sun, and Xian Zhang. A reasoning method for timed CSP based on constraint solving. In *Proceedings of the 8th international conference on Formal Methods and Software Engineering*, ICFEM'06, pages 342–359, Berlin, Heidelberg, 2006. Springer-Verlag.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[FAGNR12] Marc Fontaine, Faron Moller Andy Gimblett, Hoang Nga Nguyen, and Markus Roggenbach. Timed CSP simulator. In *Proceedings of the Posters & Tool demos Session, iFM 2012 & ABZ 2012*, 2012.

[Flo67a] R. W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Maths*, volume 19, pages 19–32. AMS, 1967.

[Flo67b] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[Fon10] M. Fontaine. Towards Reusable Formal Method Tools. In *AVoCS 2010*. Universität Düsseldorf, 2010.

[Fv93] M. Fränzle and B. von Karger. Proposal for a programming language core for ProCoS II. ProCoS II document [Kiel MF 11/3], Christian-Albrechts-Universität Kiel, Germany, August 1993.

[GG10] T. Göthel and S. Glesner. An Approach for Machine-Assisted Verification of Timed CSP Specifications. *Innovations in Systems and Software Engineering - A NASA Journal*, 2010.

[Gle05] S. Glesner. An Introduction to (Co)Algebras and (Co)Induction and their Application to the Semantics of Programming Languages. Technical Report 2005-22, August 2005.

[Gor75] Gerald Arthur Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, Dept. of Computer Science,University of Toronto, 1975.

[Gor00] Mike Gordon. From LCF to HOL: a short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.

[Göt12] Thomas Göthel. *Mechanical Verification of Parameterized Real-Time Systems*. Südwestdeutscher Verlag für Hochschulschriften AG Co. KG, 2012.

[GRA05] M. Goldsmith, B. Roscoe, and P. Armstrong. Failures-Divergence Refinement - FDR2 User Manual. http://www.fsel.com/fdr2_manual.html, 2005.

[Hay02] Ian J. Hayes. The real-time refinement calculus: A foundation for machine-independent real-time programming. In *Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets*, ICATPN '02, pages 44–58, London, UK, UK, 2002. Springer-Verlag.

[HH98] Jifeng He and C. A. R. Hoare. Unifying theories of programming. In *Relational Methods in Computer Science (RelMiCS)*, pages 97–99, 1998.

[HHMO+96] Jifeng He, C. A. R. Hoare, Markus Müller-Olm, Ernst-Rüdiger Olderog, Michael Schenke, Michael R. Hansen, Anders P. Ravn, and Hans Rischel. The ProCoS Approach to the Design of Real-Time Systems: Linking Different Formalisms. Project Report, 1996.

[HO02] Jochen Hoenicke and Ernst-Rüdiger Olderog. CSP-OZ-DC: a combination of specification techniques for processes, data and time. *Nordic J. of Computing*, 9(4):301–334, December 2002.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[Hoo94] Jozef Hooman. Extending Hoare Logic to Real-Time. *Formal Asp. Comput.*, 6(6A):801–826, 1994.

[Hoo98a] Jozef Hooman. Compositional verification of real-time applications. In *in Proc. Compositionality - The Significant Difference*, pages 276–300. Springer, 1998.

[Hoo98b] Jozef Hooman. Developing proof rules for distributed real-time systems with PVS. In *In Proceedings of the Workshop on Tool Support for System Development and Verification*, pages 120–139. Shaker Verlag, 1998.

[IR05] Y. Isobe and M. Roggenbach. A Generic Theorem Prover of CSP Refinement. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–123. Springer, 2005.

[JBK13] Jonas B. Jensen, Nick Benton, and Andrew Kennedy. High-level separation logic for low-level code. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 301–314, New York, NY, USA, 2013. ACM.

[Jon83] Cliff B. Jones. Specification and Design of (Parallel) Programs. In *IFIP Congress*, pages 321–332, 1983.

[JR97] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

[JST+09] Szilárd Jaskó, Gyula Simon, Katalin Tarnay, Tibor Dulai, and Dániel Muhi. CSP-based modelling for self-adaptive applications. In *Infocommunications Journal*, volume LVIV, pages 14–21, 2009.

[KBG+11] M. Kleine, B. Bartels, T. Göthel, S. Helke, and D. Prenzel. LLVM2CSP: Extracting CSP Models from Concurrent Programs. In *Third NASA Formal Methods Symposium*, 2011.

[KH09] M. Kleine and S. Helke. Low Level Code Verification Based on CSP Models. In M. Oliveira and J. Woodcock, editors, *Brazilian Symposium on Formal Methods (SBMF 2009)*, pages 266–281. Springer, August 2009.

[Kle11] Moritz Kleine. *CSP as a Coordination Language*. PhD thesis, TU Berlin, 2011.

[KSKA09] Anna Kosek, Aly Syed, Jon Kerridge, and Alistair Armitage. A dynamic connection capability for pervasive adaptive environments using JCSP. In *Artificial Intelligence and Simulation of Behavio ur (AISB)*, 2009.

[LA04] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–85, Palo Alto, California, Mar 2004. IEEE Computer Society.

[LA08] C. Lattner and V. Adve. LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html, 2008.

[LF08] M. Leuschel and M. Fontaine. Probing the Depths of CSP-M: A new FDR-compliant Validation Tool. In *International Conference on Formal Engineering Methods*, pages 278–297. Springer, 2008.

[LMC01] M. Leuschel, T. Massart, and A. Currie. How to make FDR Spin: LTL model checking of CSP using Refinement. In P. Zave J.N. Oliviera, editor, *FME 2001: Formal Methods for Increasing Software Productivity*. Springer-Verlag, March 2001.

[Low08] Gavin Lowe. Specification of communicating processes: temporal logic versus refusals-based refinement. *Form. Asp. Comput.*, 20(3):277–294, 2008.

[LV06] Gerald Lüttgen and Walter Vogler. Bisimulation on speed: a unified approach. *Theor. Comput. Sci.*, 360(1):209–227, 2006.

[MBK06] S. Montenegro, K. Briess, and H. Kayal. Dependable Software (BOSS) for the BEESAT Pico Satellite. In *DASIA 2006*, Data Systems In Aerospace - DASIA 2006, May 2006, Berlin. ESTEC, 2006.

[MD98] B. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: an introduction to TCOZ. In *Proc. (20th) International Conference on Software Engineering*, pages 95–104, 19–25 April 1998.

[MG07] Magnus O. Myreen and Michael J. C. Gordon. Hoare logic for realistically modelled machine code. In *In Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007), LNCS*, pages 568–582. Springer-Verlag, 2007.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *In Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996.

[Mor88] Carroll Morgan. The specification statement. *ACM Trans. Pro-*

gram. Lang. Syst., 10(3):403–419, July 1988.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[Nip02] Tobias Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In *COMPUTER SCIENCE LOGIC (CSL 2002), VOLUME 2471 OF LNCS*, pages 103–119. Springer, 2002.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Old83] Ernst-Rüdiger Olderog. On the notion of expressiveness and the rule of adaptation. *Theoretical Computer Science*, 24(3):337 – 347, 1983.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. Pvs: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer Berlin Heidelberg, 1992.

[ORY01] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. *Lecture Notes in Computer Science*, 2142, 2001.

[Pau96] Lawrence C. Paulson. *ML for the working programmer (2nd ed.)*. Cambridge University Press, New York, NY, USA, 1996.

[PL08] D. Plagge and M. Leuschel. Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more. *STTT*, 2008.

[Rey98] John C. Reynolds. *Theories of programming languages*. Cambridge University Press, 1998.

[Ros05] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 2005.

[Ros10] A.W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[Saa06] Uustalu T Saabas, A. Compiling total correctness proofs (extended abstract). In *Proceedings of the 18th Nordic Workshop on Programming Theory (NWPT'06)*, 2006.

[Sch92] S. A. Schneider. An Operational Semantics for Timed CSP. In *Proceedings Chalmers Workshop on Concurrency, 1991*, pages 428–456. Report PMG-R63, Chalmers University of Technology and University of Göteborg, 1992.

[Sch94] M. Schenke. A timed specification language for concurrent reactive systems. In D. J. Andrews, J. F. Groote, and C. A. Middelburg, editors, *Semantics of Specification Languages*, Workshops in Computing, pages 152–167. Springer-Verlag, 1994.

[Sch97] Thomas Schreiber. Auxiliary variables and recursive procedures. In *In TAPSOFT '97, volume 1214 of LNCS*, pages 697–711. Springer-Verlag, 1997.

[Sch99] S. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[Sch02] Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2,3):111–126, August 2002.

[Sch05] Norbert Schirmer. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In *Logic for Programming, AI, and Reasoning, volume 3452 of LNAI*, pages 398–414. Springer, 2005.

[Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[Sei95] Karen Seidel. Probabilistic communicating processes. *Theor. Comput. Sci.*, 152(2):219–249, December 1995.

[SST06] Klaus Schneider, Tobias Schuele, and Mario Trapp. Verifying the adaptation behavior of embedded systems. In *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, SEAMS '06, pages 16–22, New York, NY, USA, 2006. ACM.

[SU05] Ando Saabas and Tarmo Uustalu. A compositional natural semantics and Hoare logic for low-level languages. pages 151–168. Elsevier, 2005.

[Sü99] Carsten Sühl. RT-Z: An integration of Z and timed CSP. In Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors, *IFM'99*, pages 29–48. Springer London, 1999.

[TN13] Markus Wenzel Tobias Nipkow, Lawrence C. Paulson. *A Proof Assistant for Higher-Order Logic*. 2013.

[WBP06] P.T. Welch, F.R.M. Barnes, and F.A.C. Polack. Communicating complex systems. In *11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2006).*, page 11 pp., 0-0 2006.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated De-*

duction, Montreal, Canada, August 2-7, 2009. Proceedings, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.

[Wel98] P.H. Welch. Java Threads in the Light of occam/CSP. In P.H.Welch and A.W.P.Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications*, volume 52 of *Concurrent Systems Engineering Series*, pages 259–284, Amsterdam, April 1998. WoTUG, IOS Press.

[Zwi89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness*. Springer-Verlag New York, Inc., New York, NY, USA, 1989.

Publications by Björn Bartels

[Bar11] B. Bartels. Verification of Low-Level Real-Time Programs using Timed CSP. In *Formal Methods 2011 Doctoral Symposium*. University of Limerick, 2011.

[BG10] B. Bartels and S. Glesner. Formal Modeling and Verification of Low-Level Software Programs. In *10th International Conference on Qualtiy Software (QSIC 2010)*. IEEE Computer Society, July 2010.

[BG11] B. Bartels and S. Glesner. Verification of Distributed Embedded Real-Time Systems and Their Low-Level Implementations using Timed CSP. In *The 18th Asia Pacific Software Engineering Conference (APSEC 2011)*. IEEE Computer Society, 2011.

[BGG10] Björn Bartels, Sabine Glesner, and Thomas Göthel. Model transformations to mitigate the semantic gap in embedded systems verification. In *International Colloquium on Graph and Model Transformation – on the occasion of the 65th birthday of Hartmut Ehrig*, 2010.

[BJ14] B. Bartels and N. Jähnig. Mechanized, Compositional Verification of Low-Level Code. In *Sixth NASA Formal Methods Symposium (NFM 2014)*, volume 8430 of *LNCS*, pages 98–112. Springer, 2014.

[BK11] B. Bartels and M. Kleine. A CSP-based Framework for the Specification, Verification and Implemenation of Adaptive Systems. In *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011)*. ACM, 2011.

[GBGK10] S. Glesner, B. Bartels, T. Göthel, and M. Kleine. The VATES-Diamond as a Verifier's Best Friend. In Simon Siegler and Nathan Wasser, editors, *Verification, Induction, Termination Analysis*, volume 6463 of *Lecture Notes in Computer Science*, pages 81–101. Springer, 2010.

[KBG+11] M. Kleine, B. Bartels, T. Göthel, S. Helke, and D. Prenzel. LLVM2CSP: Extracting CSP Models from Concurrent Programs. In *Third NASA Formal Methods Symposium*, 2011.

[KBGG09] M. Kleine, B. Bartels, T. Göthel, and S. Glesner. Verifying the Implementation of an Operating System Scheduler. In *3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE '09)*, Tianjin, China, July 2009. IEEE Computer Society Press.