Dissertation

Combining Specification Methods for Distributed Systems

Martin Huber

Combining Specification Methods for Distributed Systems

Zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
der Fakultät für Informatik
der Universität Karlsruhe (Technische Hochschule)

genehmigte

Dissertation

von

Martin Huber

aus Karlsruhe

Tag der mündlichen Prüfung: 4. Mai 1999
Gutachter: Prof. Dr. Peter H. Schmitt
Prof. Hans Rischel

Zusammenfassung

Diese Dissertation faßt die theoretischen Ergebnisse eines Forschungsprojektes bei Siemens ZT zusammen, das sich zum Ziel gesetzt hat, die Anwendung formaler Techniken beim Entwurf verteilter Systeme zu unterstützen.

Den Ausgangspunkt der Untersuchungen bildete die Spezifikationssprache UNITY, die jedoch um zusätzliche Sprach- und Strukturierungskonzepte erweitert werden mußte, um realistische industrielle Fallstudien bearbeiten zu können. Die Erweiterungen erlauben die Beschreibung verteilter Systeme als Menge geeignet instanziierter Module, die sowohl synchron über Aktionen als auch asynchron über gemeinsame Variablen miteinander kommunizieren können. Die Spezifikationen können in einer an UNITY angelehnten Notation, aber auch als ω-Automaten oder in einer linearen temporalen Logik dargestellt werden.

Die Auswahl dieser drei Spezifikationsmethoden erfolgte aufgrund ihrer sich ergänzenden Vorteile: Die textuelle Notation erlaubt es, große Spezifikationen knapp und intuitiv zu beschreiben. Automaten eignen sich am besten zum Anschluß an Simulations- und Verifikationswerkzeuge. Sind die Automaten endlich, so können Eigenschaften mittels Model Checking nachgewiesen werden, und für überschaubare Spezifikationen bieten sie zusätzlich die Möglichkeit zur Visualisierung. Eine Darstellung in Logik erlaubt es, sowohl die Spezifikationen als auch deren Eigenschaften in einem einheitlichen Formalismus zu beschreiben.

Hauptziel dieser Arbeit ist es, formal nachzuweisen, daß diese unterschiedlichen Spezifikationsmethoden parallel eingesetzt werden können. Hierzu ist zunächst eine gewisse Vereinheitlichung notwendig, um die elementaren Sprachmittel in allen Methoden unmittelbar ausdrücken zu können. Dies wurde durch geeignete Erweiterungen der einzelnen Methoden erreicht. So ist es etwa möglich, Umgebungsannahmen direkt in der textuellen Notation zu beschreiben, Fairness für Automaten zu definieren, und synchrone Aktionen in der Logik auszudrücken. Spezifikationen in der textuellen Notation lassen sich sowohl in die erweiterten Automaten als auch in die erweiterte Logik übersetzen. Für beide Übersetzungen kann nun nachgewiesen werden, daß sie das gleiche beobachtbare Verhalten in Form von Traces besitzen.

Neben der Möglichkeit, Spezifikationen modular und hierarchisch zu beschreiben, erwiesen sich auch eine weitgehende Parametrierung und die Extraktion endlicher Modelle als notwendig, um mit realistischen Beispielen umgehen zu können. Beispielsweise läßt sich die Anzahl von Modulen eines Typs in einem System parametrieren, und die Extraktion endlicher Modelle erfolgt direkt aus der textuellen Notation.

Automatisierungssysteme dienen als typisches Beispiel für verteilte Systeme. Sie sind aufgebaut aus der traditionell datenfluß-orientierten Basisautomatisierung und der ereignis-orientierten Leitebene. Anhand einer Fallstudie wird abschließend gezeigt, daß sich die vorgeschlagene Spezifikationssprache gleichermaßen für die Basisautomatisierung und für die Leitebene eignet.

Abstract

This thesis summarizes the theoretical results of a research project at Siemens ZT that aimed at supporting the use of formal methods in the design of distributed systems.

The specification language UNITY formed the starting point of the investigations. However, UNITY had to be extended with additional language and structuring concepts in order to deal with realistic industrial case studies. These extensions allow the description of distributed systems as set of properly instantiated modules that communicate using both synchronous actions and shared variables. The specifications may be represented in a UNITY-like notation, but also as ω-automata or in linear temporal logic.

The choice of these three specification methods resulted from the observation of their supplementing advantages: the textual notation allows the concise and intuitive description of large systems. Automata are best suited for connecting simulation and verification tools. If the automata are finite then properties may be proven using model checking, and the automata may serve as visualization. The logical representation enables the user to describe both the specifications and their properties in a uniform formalism.

The main goal of the thesis is to formally demonstrate that these specification methods may be used in parallel. In order to reach this goal a standardization proved to be useful in order to be able to directly express the elementary language constructs in all methods. This is achieved by appropriate extensions for the methods. For example, it is possible to describe assumptions concerning the environment directly in the textual notation, as well as to express fairness for automata and synchronous actions in the logic. Textual specifications can be translated to both the extended automata and the extended logic. Both translations can be proven to result in the same set of observable behavior in terms of traces.

In addition to modular and hierarchical specifications, two concepts showed to be useful to deal with realistic examples: parameterization and the extraction of finite models directly from the textual notation.

Control systems serve as a typical example for distributed systems. They are built up from the base automation that traditionally is dataflow-oriented and the event-oriented high level process control. By means of a case study it is finally shown, that the proposed specification language is suited equally for the base automation and the high level process control.

Ich hatte das Glück, zur Erstellung dieser Arbeit mit einem Doktorandenstipendium der Siemens AG unterstützt zu werden, für dessen Gewährung ich Herrn Prof. Büttner meinen herzlichen Dank aussprechen möchte. In der Fachabteilung ZT SE 4 fand ich eine fachlich anregende Arbeitsumgebung, in der ich mich sehr wohl gefühlt habe. Neben meinen Münchnern und Erlangern Kollegen danke ich dabei auch den Gastwissenschaftlern und Werkstudenten, die das Projekt TLT begleitet haben.

Den Gutachtern, Herrn Prof. Schmitt und Herrn Prof. Rischel, danke ich für das meiner Arbeit entgegengebrachte Interesse und für die hilfreichen Anregungen.

Zu guter Letzt von ganzem Herzen ein Küßchen für Karla.

Contents

Chapter 1

Introduction

1.1 Motivation

In the past decade a lot of research was dedicated to distributed systems, which emerge in such distinct areas as, for example, airline reservation systems, avionic systems, operating systems, or process control systems. All these systems have in common that they consist of a number of (often even physically) distributed components that in order to fulfill their common task have to be coordinated and therefore communicate from time to time with each other. Typically, at least some part of a distributed system shows some ongoing behavior and mostly, distributed systems are nondeterministic, either because the systems themselves are nondeterministic or because they have to react to the inputs of some more or less unknown environment. These two aspects, the necessity of interaction and the consideration of infinite and nondeterministic behavior, make the specification and realization even of small distributed systems a challenging and often enough error-prone task.

On the other hand, many distributed systems are used in safety critical areas where the cost of a failure may be immense. Thus, much effort is spent to assure a high reliability. Traditionally, this is done in late development phases by reviewing or testing the executable code. However, in recent years these methods were more and more considered both insufficient and inefficient, that is expensive. At the same time, the emergence of efficient data structures and algorithms for examing in a rigorous way mathematical models of such systems as well as the emergence of temporal logics to describe properties of these systems intuitively but also mathematically exact, raised hopes of applying these so-called formal methods to industrial size distributed systems.

Formal methods seemingly fit better to the specification of systems in early design phases because less details in the description of the system have to be considered but also because in later design phases the description of the original system intertwines with implementation details. As a consequence, for a formal analysis in later design phases both the system and the programming language it is described in or, even worse, the hardware it is supposed to be executed on, have to be modeled. Thus, rather naturally, formal specification and verification of systems complements traditional reviewing and testing of code.

This thesis deals mainly with such a kind of formal specification of distributed systems. As formal specification, or specification for short, any description of a distributed system will be considered that has some well-defined semantics in terms of a set of observable ongoing behavior. Also dealt, but of minor concern, are the verification and the realization of distributed systems. The main focus hereby is on so-called reactive (embedded) systems that react to (are embedded

in) an environment that is at least partially unspecified and in worst case can issue any allowed input at any time.

In this thesis, as well as in most other formalisms there is a distinction between a system description (or "programming") language and a property language with the tendency of the system description language being more operational and the property language being more descriptive. Nevertheless, both languages are tied together closely, due to the fact that both a system and a property manifest themselves in a set of observable behavior. In fact, a specification viewed as a set of behavior, may be equally considered a description of a system itself or a description of a property of that system. Therefore, in the sequel, I use the term specification language to include both a system description language and a property language.

In order to deal with large distributed systems, a specification language should

1. **be comfortable but not too complex**
2. **be compositional**
3. **support different abstraction levels**
4. **offer tool support**

The first item is guaranteed in the approach advocated in this thesis by using a small but powerful set of basic concepts:

- specifications are structured hierarchically in three levels, namely blocks, modules and systems (comparable to the hierarchy functions/procedures, processes, and systems found in operating systems),
- there are two concepts for communication: shared variables and actions, the latter being used for synchronous communication,
- there are two concepts for describing transitions by distinguishing which transitions must be executed and which ones may be executed,
- infinite behavior may be influenced (only) by weak and strong fairness conditions, and
- restrictions on the environment are explicitly stated in the specifications as assumptions.

The second item, that is, the demand for compositionality, may be found as a requirement in nearly every paper on the specification of distributed systems. Unfortunately, there are many different notions of compositionality. In this thesis, a specification language is called compositional, if well-defined units of a specification are fully abstract , that is, if they have the same semantics (meaning) in any context they may occur in. The important thing here is, that some fully abstract block or module may be understood (but also specified, refined and verified) completely "stand-alone". More formally, there is a function **Traces** that for each such unit U of some specification uniquely determines $\mathbf{Traces}(U)$, the set of observables behavior of U. If two units U_1 and U_2 get composed, noted by $U_1 \| U_2$, then by definition of $\|$ it is guaranteed that the set of observable behavior corresponding to the composition is simply the intersection of the two sets of observable behavior corresponding to the components, that is, $\mathbf{Traces}(U_1 \| U_2) = \mathbf{Traces}(U_1) \cap \mathbf{Traces}(U_2)$. Furthermore, the composition is only defined if the existence of at least one overall behavior is guaranteed, that is, $\mathbf{Traces}(U_1 \| U_2) \neq \emptyset$ by construction. This is achieved mainly by assumption commitment reasoning.

Thirdly, the specification language should allow to write both very abstract specifications with many details left out or specified non-deterministically, but as well specifications that are detailed

enough to be compiled to some executable program. This frees the user of learning a variety of different languages and is the foundation for applying a uniform refinement method all over the development process. Indeed, this uniform refinement relation between specifications S and T is simply the set inclusion relation on the sets of observable behavior, that is, T refines S if and only if $\mathbf{Traces}(T) \subseteq \mathbf{Traces}(S)$, expressing that every observable behavior of T is contained in (and thus is described by) the set of observable behavior of S. Vice versa, in this situation, S is called an abstraction of T. Note again, that it does not matter whether S and T both are properties or both describe systems or some mixed situation arises: It is typical that in early development stages both S and T are properties, whereas in later development phases at least T describes a system in some programming notation. Yet another situation occurs, if some property P has to be shown for some given system description S, and S turns out to be "too big" or can not at all be finitely represented. In this situation, one looks for some finite representation T such that by construction $\mathbf{Traces}(S) \subseteq \mathbf{Traces}(T)$ and furthermore $\mathbf{Traces}(T) \subseteq \mathbf{Traces}(P)$ can be proven by some decision procedure.

Tool support useful for specifying large distributed systems includes editors, organization tools for documentation and version control, simulation and verification tools. Although the tool box may contain both automata and logic based verification tools, for the user of these tools both the programming notation and the property language should be uniform.

1.2 Project Description and Development

This thesis was developed at the research department ZT SE 4 of Siemens at Munich. The main focus of ZT SE 4 is the support of the engineering process within Siemens. This support consists of applying modern simulation, verification, optimization, code generation and synthesis techniques to industrial problems. Many examples deal with distributed and reactive systems like, for example, communication systems or train control systems.

To *formally* specify these various distributed systems, the TLT project started in 1991 by analyzing UNITY as introduced in [CM88] . UNITY consists of a programming notation and a simple yet expressive logic to reason about the programs as well as a refinement methodology for stepwise development of distributed systems. The main drawbacks of the original UNITY approach are that it is not really compositional (see [San91] or [UK92]), and there is no means to directly express synchronization. My own focus at that time was refinement (in [Hub93] I formalized some refinement relations methodology for a UNITY-like language that, for finite state programs, could be checked automatically by model checking techniques).

UNITY specifications are not structured in fully abstract units. This complicates the handling especially for larger examples. This lead us to considering interfaces of specifications, a concept well-known by software engineers (see [AG94]). The interfaces should hold all relevant information about any admissible environment of the specification. One such information defined in an interface is the *environment class* of variables that determines, for example, whether a module has write access for a variable. Under the guidance of project leader Jorge Cuellar, a first comprehensive language definition for TLT emerged that in essence extended UNITY by interfaces. Gerd Gouverneur, Dieter Barnard and I translated specifications of this language to an automata format (coded as binary decision diagrams as introduced in [Bry86] and [Bry92]) that could be interpreted (stepwise executed) and verified using the decision procedures of Thomas Filkorn's model checking tool SVE (see [FSS+94]). Some verification is done implicitly, as, for example, checking whether there is some legal next state for every guarded command.

Additional properties can explicitly be stated in a UNITY-like logic (see [Hub93]).

In 1994, Holger Busch with a strong background in higher order theorem proving joined the group and at the same time we started some joined work with Stephan Merz who just finished a year of work with Lesli Lamport on TLA. This lead to a second line of tools by translating TLT specifications to TLA (see [Mer95]) for which Holger Busch implemented a shallow embedding in the higher order logic theorem prover Lambda (see [Bus95] and [Bus96]). At around the same time, Jorge Cuellar, Dieter Barnard and I worked on synchronization which we wanted to include into the language as a means to write more abstract specifications. After comparing the synchronization concepts of CCS (see [Mil89]) , I/O-automata (for an introduction see [LT89]) and in the S/R model of Kurshan (see [Kur94]), we decided to use "directed" actions as a synchronization concept in the programming notation, whereas in the logic and automata-based translations synchronization is not directed.

Together with two students, Christine Röckl and Dagmar Pröll, I implemented a compiler for TLT that compiles TLT systems to distributed C-programs using the MPI (message passing interface) library. MPI also allows to visualize the execution of the resulting C-programs (see [CHB96b]).

In [Bar97], Dieter Barnard presents a version of TLT specialized on client-server systems. Further, he describes the tools mentioned above.

TLT is a research project whose goal it is to support development projects. As a main focus of our department is on reactive process control systems, TLT has been applied to several of these systems, see [CWB94, CH95, CW96]. Technology transfer further took place to extend the CSL language (for Control Specification Language) to also deal with distributed systems, resulting in the language definition of DCSL (for Distributed Control Specification Language).

Besides, TLT has been applied to distributed train control systems, an ATM signalling protocol (see [BC95]), and to the RPC-memory specification problem of Manfred Broy and Lesli Lamport (see [CHB96a]).

1.3 Contribution

Summarized, the TLT project has arrived at a situation where specifications arise in three different "views" or "representations":

1. A textual, operational one suited for example for some compiler but also the most comfortable (readable) representation for the developer.
2. A logical one, suited to reason about the specifications in some logical calculus (by hand or with assistance of a theorem prover). The same logic is further used to express properties of specifications.
3. An automata based one in form of a transition system labeled by elements of a Boolean algebra of appropriate FOL predicates, suited best for decision procedures like model checking.

To be able to safely mix these representations, they have to be proved "equivalent" which is the main concern of this thesis.

The unification of the logical and the automata based view will be done by comparing the sets of traces (the sequences of observable behavior) of both views and show them to be equal. For

the operational representation one might define a structural operational semantics and map configurations to traces. Instead, the textual representation is considered as given and is used to extract in essence a transition relation that forms the core of both the logical and automata based representation used by the tools (see Figure 1.1). Being freed of syntactic sugar, the two intermediate representation (in logic or as automata) are suited to be handled by tools and for formalizing general theorems or methods dealing with specifications. Some translation back from the (flat) logical or automata based view to the (more structured) textual representation is not considered (anyway it would be quite cumbersome to obtain "readable" results).

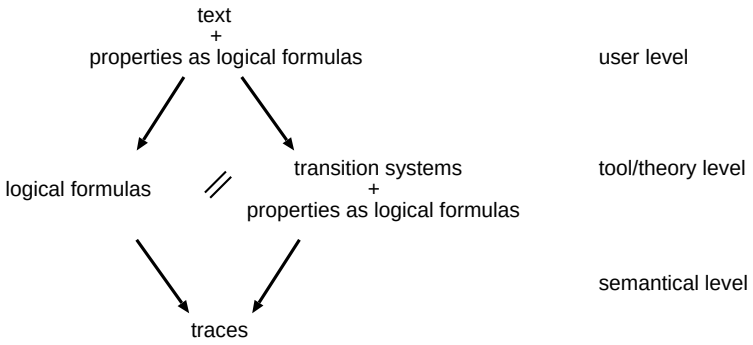


Figure 1.1: The connection between the different views of a specification.

In this thesis, I allow both communication based on shared variables and communication based on actions and I show that the traces of the automata-based models of a specification agree with the traces of temporal logic formulas corresponding to that specification. By this, and by the fact that there is only one common notion of composition, namely trace intersection, it is guaranteed that tools from the two different worlds can be safely used together in the verification of one system. In the more academic literature it is often argued that one communication mechanism is enough, as other ones can be mimiced more or less smoothly. This way, formalisms like UNITY and TLA (only shared variables) or CCS and I/O-automata (only synchronization), are kept elegant. However, in my view, many specifications are much more natural and also more concise if both shared variables and actions may be used directly. Thus, on this point, I spent some additional effort on the language definition in favor of the conciseness of the specifications to be noted in that language.

In order to realize a more direct correspondence between the programming notation and the temporal logic, I include actions representing synchronous channels into first-order logic and into linear time temporal logic presented in Chapter 3. Although only minor changes to the logics have to be made, such an extension (to my knowledge) has not been considered prior to TLT.

In Chapter 4, Boolean transition systems are introduced. As acceptance conditions I mainly use fairness conditions. Further I present a construction which mechanically transforms Boolean transition systems with strong fairness conditions to Boolean transition systems with only weak fairness conditions. The use of this construction is twofold: the weak fairness conditions result in faster decision procedures, and weak fairness conditions may be expressed as Büchi-conditions that are compositional. From I/O-automata (see [LT89]), I employ the notion of input enabledness but at the same time I introduce delays as a new, less restrictive concept that also allows non-blocking synchronous composition.

When talking to engineers, it is often difficult to extract a useful set of properties that guarantee important properties of a system. In this thesis, I present consistency criteria, that are strong enough to guarantee for example freedom of deadlock but also that there is an implementation (a model) of the system.

Inspired by [HL95], I also deal with abstract Boolean transition systems. As a good first guess, I present a construction that extracts the control information from the textual representation and uses this information to build a finite state Boolean transition system.

Some of the material of this thesis has been presented before. Not published previously are

(1) a version of TLT that allows to mix guarded and triggered transitions within one block (see Section 5.7),
(2) the notion of delays (see sections 4.2.6 and 5.6.3),
(3) the extraction of symbolic Boolean transition systems from the text (see Section 5.7.3),
(4) the algorithm to replace strong fairness conditions (see Section 4.2.3) , and
(5) any of the proofs.

1.4 Related Work

A huge variety of specification languages for distributed systems emerged over the past decades. For a more complete overview, the reader is referred to [Ber93]. In the sequel, I only consider the languages that influenced this thesis.

UNITY , as introduced by Chandy and Jayadev Misra in [CM88], consists of a programming notation and a linear temporal logic used both for expressing abstract specifications and for reasoning about programs. UNITY programs consist of a static, finite set of statements that may change the values of a set of program variables by means of so-called multiple assignments. This means that all assignments of one statement are executed simultaneously. An execution of a program starts in some specified initial state and thereafter continues to choose and execute statements. The choice of the next statement is arbitrary besides the fairness rule that each statement is chosen infinitely often.

Besides the common **unless** and **leadsto** operators, the UNITY logic further contains **ensures**, an operator that is directly linked to a program statement. This, often called helpful statement, makes sure that the progress expressed by the **ensures**-property actually occurs. In [CH93] and [Kal96] this idea is extended to chains of helpful statements (in [CH93] chains of sets of helpful statements) and thereby represents a kind of "guided leadsto".

UNITY is so appealing mainly because there is nothing more to be said about its fundamental language constructs. Nevertheless, it is commonly accepted that UNITY lacks further structure (for example, UNITY has no notion of hiding) and some effort was spent to add local variables (see [UHK94]), compositional logic operators (see [San91]), or rely/guarantee-style reasoning (see [CK93]).

Recently, Jayadev Misra introduced Seuss, a language that structures specifications in boxes that contain total procedures for the transformational aspects of a specification and partial procedures for reactive aspects. The only communication mechanism of Seuss is procedure call. This makes it possible to understand a program execution as a single thread of control, despite possible concurrent implementations (see [Mis96]).

TLA (the temporal logic of actions, see [Lam94a]) does not distinguish between a program notation and a property language. Instead, TLA specifications consist solely of specialized linear temporal formulas. One big advantage of this approach is that composition thereby is reduced to conjunction. The foundation of these formulas are so-called actions. These are first order logic formulas referring to a set of program variables as well as to primed copies of those variables. The primed variables are used for the values of the corresponding program variable in the next step of an ongoing behavior. For example, $x' = x + 1 \wedge y' = y + 1$ indicates that x and y are increased by one what would be noted $x, y := x + 1, y + 1$ in UNITY. Using predicates allows specifying arbitrary relations like $x' > x$ with the intended meaning that x is increased by some unspecified positive value.

Lamport advocates using a special notation of a linear temporal logic based on actions, the **always** operator, and weak and strong fairness. Using this notation facilitates to denote so-called stutter invariant formulas.This allows a powerful refinement methodology whose basics may be found in [AL88]. In TLA, refinement is expressed in essence by simple implication. For example, $\Phi \Rightarrow \exists_a \Psi$ means that specification Φ refines specification Ψ, or, more formally, that for every behavior fulfilling Φ one may find sequences of values for the abstract variables a such that Ψ is fulfilled. The high expressiveness of this refinement construct is due to the existential quantification over behavior.

As in the case of UNITY, lacking structuring facilities were added subsequently (see [Lam94b]).

The Calculus of Communicating Systems, or CCS for short, was introduced by Robin Milner in [Mil80] and slightly revised in [Mil89]. In CCS, specifications are given as algebraic expressions. For example, $a.E$ denotes a process that takes part in an action a and thereafter behaves like process E. In CCS, an action a may happen simultaneously with the action $\bar{a}$ of another process, resulting in the composed system in the unobservable internal τ-action. CCS specifications may be represented as labeled transition systems and they may be given a trace semantics. However, with respect to this trace semantics, CCS is not compositional, since the traces do not adequately represent deadlock situations of a system.

From CCS, TLT only incorporates the idea of synchronous communication that is also central to the synchronous languages Esterel, Lustre, and Signal developed at INRIA (see [BB91], [Hal93]).

I/O-automata are state-machines with state transitions labeled by actions that are classified as either input, output, or internal actions. As in CCS, the visible (that is, input and output) actions are used for synchronization. In contrast to CCS or CSP (Communicating Sequential Processes, see [Hoa85]), every automaton must accept any of its input actions at any time. If systems are composed of several automata, then each action is declared at most once as an output action. Thus all but at most one component are passive recipients of any action. As a consequence, a system composed of several I/O-automata may never deadlock. Furthermore, this has the advantage that, in contrast to CCS or CSP,the trace semantics of I/O-systems is compositional. Another difference to CCS is that matching pairs of output and input actions are not hidden in the composed system but still are considered to be output actions. As in TLA, fairness is used to restrict the infinite behavior. For describing transitions, a simple textual schema is used, splitting transitions in their precondition and their effects respectively.

In the S/R model (S/R stands for selection/resolution) as presented in [GK80] or [Kur94], processes are described as labeled transition systems. A shared memory serves as synchronization means. Each process may read any shared variable but has write access only to a subset of the

shared variables, called its selection variables. The sets of selection variables are distinct for each process. A computation step now consists of a selection phase followed by a resolution phase. In the selection phase, all processes nondeterministically choose, depending on their current local state, values for their selection variables. Hereby, the set of possible values is given as a predicate labeling each local state. The global selection, made up of the selections of all processes, is called the resolution of the system. Depending on the resolution, each process determines one enabled transition, that is, a transition whose label, again given as a predicate over all shared variables, is consistent with the global selection. In [CGS91] it is shown that the labeling of the local states with a selection may be omitted. The resulting structure is called Boolean transition system and formed the basis for the Boolean transition systems used in this thesis. In essence, I added stuttering and fairness to the approach of [CGS91].

Through cooperative work, TLT further was influenced by the ideas of Egon Boerger, Hans Rischel, and Catalin Roman advocating abstract state machines[1] (see [Gur91]), duration calculus (see [RCM+95]), and Swarm (see [RC90]) respectively. It can also be related to the functional approach of [Bro96], even though this relation is somewhat cumbersome [Mer96].

1.5 A Note on UML

Since the release of version 1.1 in September 1997 (see [Rat97a]), the Unified Modeling Language (UML) has quickly become a standard specification language. Although UML has not influenced the development of TLT, it is now briefly compared to TLT.

UML allows the modeling of (distributed) systems using different kinds of graphical notations. For example, class diagrams and object diagrams are used to model in an object-oriented way the static relationship between classes and objects respectively. The dynamic aspects can be specified using state, sequence, collaboration and activity diagrams. Furthermore, there are use case, component and deployment diagrams.

None of the diagrams in UML is intended to capture the complete system. Instead, the numerous diagrams illustrate different aspects of the system under investigation. In general, there is no code generation for the dynamic aspects of the system. Currently, simulation and verification are still impeded by the fact that none of the diagrams has a formal semantics (in [Rat97b], the semantics of the diagrams is given in plain text).

TLT, on the other hand, is intended to specify the whole visible behavior of a system within one framework. This is possible, because in TLT the only "associations" between the objects of a system are given by the communication mechanisms of TLT. The "multiplicity" of associations is reflected in TLT by allowing vectors of variables, actions and modules. In this sense, the modules in TLT correspond to classes in UML and a system description in TLT corresponds to an object diagram. However, the modules named in a TLT system description are not restricted to static aspects but also represent a dynamic behavior that, due to the formal semantics of TLT, often can be simulated, verified and implemented.

Using a standard specification language has a lot of advantages regarding user acceptance and tool support. Therefore, it is a current research activity at ZT SE 4 to adapt UML for control systems.

[1] Formerly, abstract state machines were called evolving algebras.

1.6 Thesis Outline

The introductory example presented in Chapter 2 introduces the main features of the TLT specification language and its methodology. This introduction provides enough information to follow the case study in Chapter 6. Besides, it is helpful to keep track of the more theoretical chapters that follow.

The following two chapters present the adapted versions of two theories widely used as models of concurrent systems. The first one, (linear) temporal logic, is suited to describe both concurrent systems and their properties. This allows deriving properties of systems in one "homogeneous" calculus. The second one, Boolean transition systems (BTS), generalizes labeled transition systems and is suited for applying decision procedures (at least given some finite state space). The aim is to allow the use of both theories within the verification of one system in a consistent way. The glue combining both models are traces. Traces are sequences of visible states and actions, defined either based on the allowed paths through a BTS or as the models fulfilling a temporal logic formula.

In Chapter 5, a programming notation for TLT is defined together with the translations to temporal logic formulas and Boolean transition systems. It is shown that both translations result in the same set of traces. Consistency criteria are given that guarantee that there is at least one trace fulfilling the specification. Furthermore, finite abstract Boolean transition systems are defined and their relation to the exact concrete models is examined.

In Chapter 6, the theory is applied to the case study "Fault Tolerant Production Cell", defined by the BMBF project KorSys.

Chapter 2

Introductory Example

The example presented in this section describes a situation that frequently arises in control systems: a group of motors has to be started and stopped in predefined orders. On the basis of this example, the main concepts of TLT are introduced to guide the reader in the more theoretical chapters that follow.

The first section in this chapter introduces the example and briefly analyses the traditional engineering process. The following two sections then deal with the engineering process in TLT and the development of modules in TLT.

2.1 The Motor Group Example

A typical control system at least consists of a process to be controlled, a controller and an operator station (see Figure 2.1). The controller reads sensor values, calculates some control function, and writes the actuator values accordingly. Often, the controller runs on specialized hardware like programmable logic controllers (PLC). The controller also notifies the operator about important changes in the process. Vice versa, the operator can influence the process by notifying the controller. The controller transforms the commands of the operator into appropriate actuator values.

The process in the motor group example consists of a row of very simple motors. The control task simply requires the motors to be started one after the other and to be stopped in reverse order. Both operations are initiated by the operator who in turn receives notifications about success or failure. Furthermore, it is assumed that some unspecified error may occur any time in the process. The occurrence of this error as well as a "StopGroup"-command from the operator station immediately cause the controller to stop the motors, even if a starting operation is not yet finished.

In the sequel, the main focus is on the controller.

2.1.1 Traditional Engineering Process

In the field of control systems, it is common to separate the development of basic building blocks from the engineering process for a concrete plant.

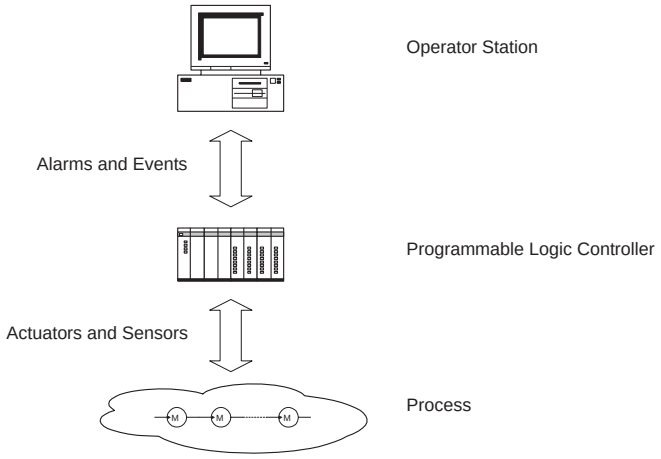


Figure 2.1: A simple control system.

The basic building blocks represent specialized control functions. Some of these blocks may be delivered in standard libraries shipped with the process control system whereas others are developed specifically for an application area. In the dominant standard IEC 1131 (see [tcN92]), these basic blocks are called function blocks. Function blocks have a uniform and well defined static interface whereas the behavior can be written in any of the IEC programming languages instruction list, ladder diagrams, function block diagrams, sequential function charts and structured text. The first three of these languages are the traditional languages whereas the latter two allow, for example, to easily specify the control flow of a program.

The engineering for a plant is then done graphically by instantiating and connecting the function blocks as shown in the continuous function chart in Figure 2.2.

The continuous function chart is built up by two instances of a function block called **Motor**. The coordination is done by the function block **GroupCtr** . Finally, there is a function block **StopCondition** that detects lock conditions. The connections between the function blocks mark the data flow. In the example, some of the inputs and outputs are connected directly to the process input and output (given as hardware addresses). In addition, the user has to determine the sequence in which the function blocks get executed. Furthermore, a mapping to the operator station and higher level process control systems has to be realized by setting attributes for the inputs and outputs of the function blocks. The programmable logic controller then cyclically reads the process inputs and operator commands, executes the program in the specified order, and writes the process outputs and operator notifications.

Although this state-of-the-art style of engineering process is much more comfortable than the pure instruction list and ladder diagram solutions used traditionally, it still has several deficiencies:

1. **Application domain**
   Continuous function charts are usually restricted to the level of the programmable logic controllers. Todays higher level control systems like batch control systems, typically run on PC's and are programmed separately using different languages and concepts.

2. **Generic designs**
   Reuse is hindered by insufficient support for generic designs: adding a third motor in the

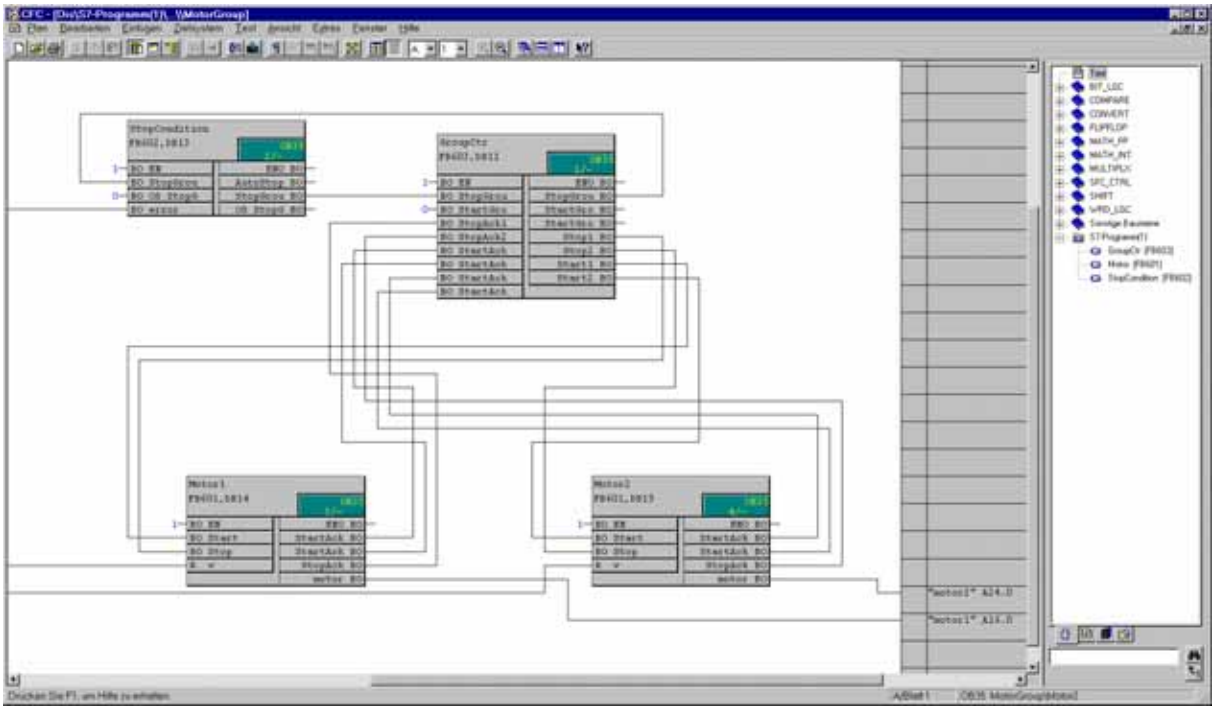


Figure 2.2: A continuous function charts for the motor group example.

example would require the user to program a new group controller simply because this function block needs more inputs and outputs.

3. **Clearness and safety**
   For larger examples, the graphs lack conciseness. Even for small plants there are hundreds of charts with thousands of function blocks and connections. The function blocks may be connected almost arbitrarily, because there is only a static type checking. Furthermore, the usefulness of the type checking is limited by the fact that more than two thirds of all variables are usually of Boolean type.

4. **Abstraction**
   Continuous function charts are not intended and not suited for high level designs. For example, there is no notion of refinement that would be necessary to describe systems at different levels of abstraction.

5. **Efficiency**
   Sometimes, it is difficult to make the control programs efficient: in the example, there is no need to execute the group controller in each cycle. Instead, it would be sufficient to trigger the controller only if one of its inputs changes its value. Because efficiency often is crucial, such situations are dealt by introducing further execution models based on alarms, timers and interrupts all executing in parallel to the cyclic execution order described above. However, this further complicates the programs and the testing.

2.2 Engineering in TLT

Although never realized, it would also be possible to use a graphical tool in TLT to describe the system level of a specification (see Figure 2.3). The main difference to the continuous function chart is that the TLT specification is not necessarily restricted to the control layer. In general, there are also modules modeling the operator station, a batch control system, or the process. A second important difference concerns the data flow. In TLT, data flow is based on both (shared) variables (shown as dashed lines) and synchronous actions (represented by solid lines). For clarity, the data types are omitted in the figure. Finally, in contrast to the solution presented before, the controller module is generic. By simply choosing a value for the parameter nbMotors, the interface and the behavior of the module can be adjusted to deal with any number of motor modules.

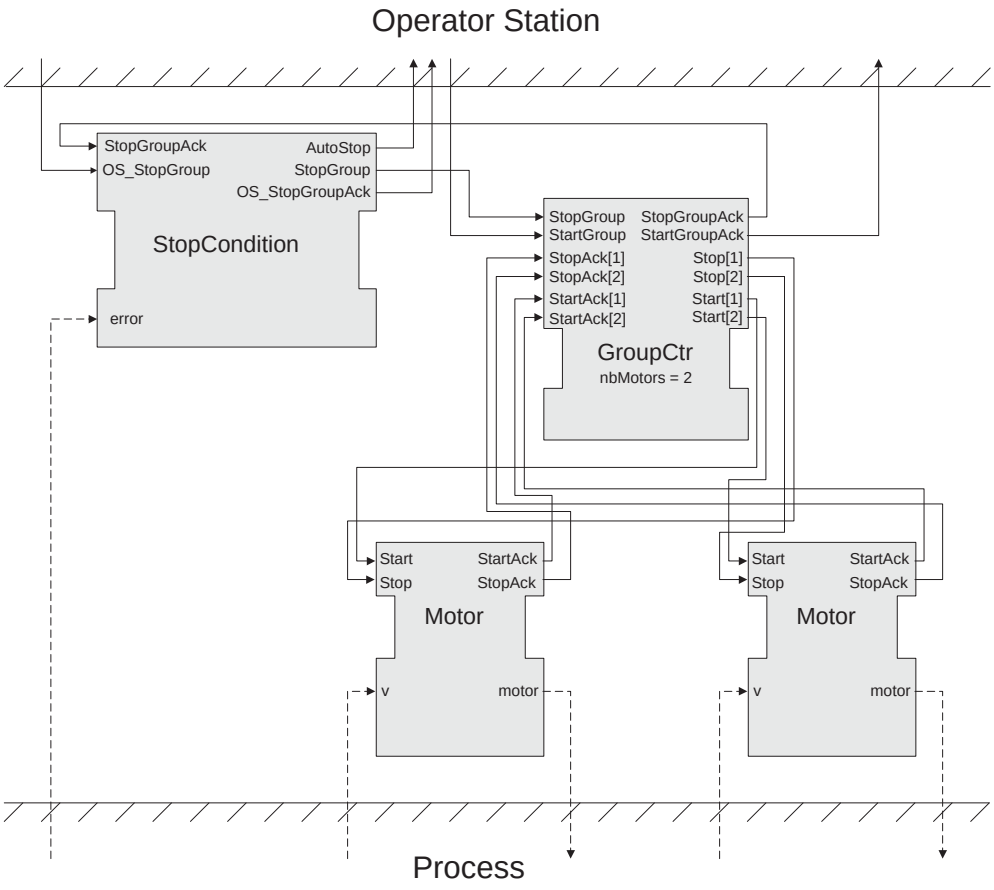


Figure 2.3: On the system level, the user instantiates and connects modules.

The following textual description is equivalent to the graphical one presented in Figure 2.3:

```
System  MotorGroup
    Parameters
        MotorNo     : Natural  := 2
    Layer 1
        Include  Module  StopCondition
        Include  Module  GroupCtr [nbMotors ← MotorNo]
        Include  ⟨ m : [1..MotorNo] ⟩  Module  Motor [ Start ← Start[m], StartAck ← StartAck[m],
                                                       Stop ← Stop[m], StopAck ← StopAck[m],
                                                       v ← v[m], motor ← motor[m]]
End
```

In the textual description, it is no effort at all to change the number of motors. One only has to change the parameter MotorNo. There is no need to draw any connections, because these are generated automatically by identifying variable and action names. Therefore, any identifiers occurring in a module can be renamed when the module gets instantiated in a system. In the example, some parameter nbMotors of the controller module gets renamed to MotorNo. Besides, the string "[m]" is appended to the variable and action identifiers[1] of any included motor module. After the renaming has been carried out, the motor module is instantiated |MotorNo| times with m taking values from the integer interval [1..MotorNo]. Altogether, this results in variable names like v[1] and action names like Start[1].

In this example, all modules are part of one layer. In general, system descriptions can consist of several layers that alternate with Property-sections. In these Property-sections, one may describe properties that are guaranteed by the lower layer(s) and on whom the upper layer(s) may rely. A Property-section for the motor group example is presented at the end of this chapter after having introduced temporal logic as the language of choice to specify the properties.

This section closes with a sequence diagram that describes one possible intended behavior of a system with two motors (see Figure 2.4).

The notes in the diagram show the conditions that must hold to cause an action as well as the side effects that actions cause in the process. For example, a Start[1]-command causes the first motor module to set the variable motor[1] that is connected to the physical motor.[2] Likewise, the module can only acknowledge the Start-command if the velocity of the corresponding motor as represented by the variable v[1] differs from 0.

2.3 Developing Blocks, Modules and Systems

In the sequel, the main features of the language are introduced bottom up, starting with blocks.

[1] By convention, action identifiers start with capital letters whereas variable identifiers start with lower-case letters.

[2] As in B, Z, or TLA, the prime is used to indicate the value of a variable after a transition took place.

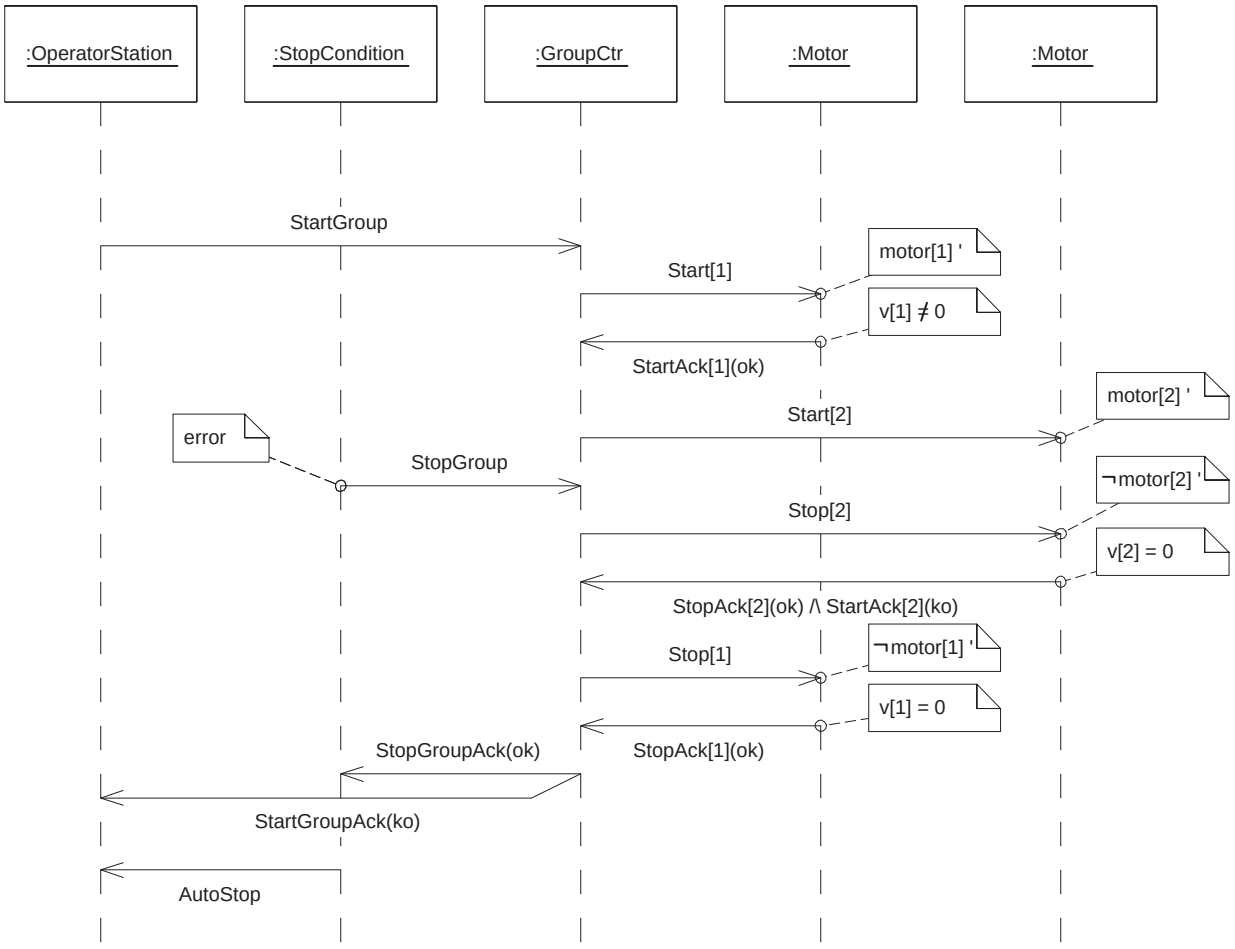


Figure 2.4: The sequence diagram shows the behavior of the system if some error occurs in the process while the motors are getting started. Due to that error, the motors get turned off in reverse order.

2.3.1 Blocks

In TLT, it is possible to further structure modules into blocks. A block is the smallest unit that may be given a semantics. Blocks "control" or "own" a set of variables and actions no other block or module is allowed to change and emit respectively. For variables this means that at most one block has write access, for actions this means that at most one block can initiate them (not controlled by any block of a given module are Read variables and In actions, see below).

Blocks used to specify a communication protocol often fulfill some syntactic restrictions that allow to use an exact "mirror"-block for the communication partner. Such blocks are called interfaces and inverting such an interface means that all incoming actions become outgoing actions and vice versa. An inverted interface describes exactly the same behavior as the original interface. Therefore, the composition of the two is trivially non-blocking.

Taking a look at the sequence diagram in Figure 2.4, it becomes obvious that all commands like Start[1] or StopGroup are intended to get acknowledged. This observation justifies the introduction of an interface that describes a simple protocol where an incoming Cmd action and

an outgoing CmdAck action alternate, starting with a command.

The textual representation of this protocol reads as follows:

```
Interface  Acknowledge
  Declarations
    Variables
      History     pending  : Boolean
    Actions
      In          Cmd      : ()
      Out         CmdAck   : { ok, ko }
  Initially    ¬pending
  Transitions
      [tr1]    {¬pending}    Cmd   =▷  pending′
      [tr2]    {pending}     CmdAck  =▷  ¬pending′
End
```

The textual view of a block starts with the keyword Block or Interface followed by the name of the block. The next sections for declaring types, parameters, and abbreviations are not necessary for the example and thus omitted here. Then follows a set of declarations of variables and actions. Besides their domain, variables (and actions) are given an environment class: Write variables belong to the module which declares them; they are visible to the environment, but they may not be modified by it. Read variables are imported from the environment; their values can be read but not modified locally. History variables record the history of visible events between a module and its environment. Furthermore, there are Spec (specification) variables that are not part of the state space of modules; they are used to define constants as well as to parameterize transitions. Local variables finally completely belong to a module; they are not visible and therefore they may neither be read nor changed by the environment. Within one module, any block may read any variable whereas only one block has write access to a variable that may be changed by the module. This block is said to "control" the variable.

Actions are used in CCS-style (see [Mil89]), that is, matching visible actions allow for synchronous communication among modules. Visible actions are therefore declared as In or Out, depending on whether they are under the control of the environment or the given module. So-called Internal actions may be used to synchronize the blocks within one module. They are not visible to the environment. Again, all blocks of a module "see" and may react to any action, but at most one block may issue an Internal or an Out action. This block is said to "control" the action. Whereas variables always hold some value, actions only "occur" occasionally. Whenever an action occurs, a value is communicated instantaneously according to the domain of the action.

The declaration of a variable or action with name Identifier is denoted according to the schema

Env_Class Identifier : Domain

where Env_Class and Domain determine the environment class and domain of the variable or action. Domains may be, for example, Boolean, integer, real as well as sets, arrays or lists.

One special domain, called unit, indicates actions that carry no value. These actions are called signals and thus merely serve for synchronization.

In the example, it is assumed that the interface is responsible for handling one particular command. Thus, the command can be modeled by a signal, that is, an action that provides no "value passing". The CmdAck action can take either value of the set { ok, ko }. To describe this protocol, one Boolean valued history variable pending is sufficient: pending is true if and only if a command is pending, that is if and only if a command already has been issued but the corresponding acknowledgement has not yet occurred.

The variables that are controlled by a block are given some initial values in the Initially section. The initial values are determined by a predicate like, for example, $0 < x \wedge x < 10$. Controlled variables not occurring in the initial section may assume arbitrary initial values from their respective domain.

The core of a block is the Transitions section, describing operationally the next step relation. Two kinds of transitions are distinguished:

1. *Guarded transitions* : [] g $\longrightarrow$ cmd
   These are essentially Dijkstra's guarded commands: if guard g is true ("enabled") then the command cmd *may* be executed. By cmd, controlled variables are updated or controlled (that is Out) actions are executed. [] is used as separator; for reference purposes the transition may have a label (for example, [gt1] g $\longrightarrow$ cmd).

2. *Triggered transitions* : [] {ac} ev=▷cmd
   The semantics of triggered transitions may informally be characterized as follows: whenever event ev occurs then the command cmd *must* be executed. An event is an observable transition either controlled by the module itself or by its environment. In this thesis, all events are Boolean combinations of actions. The (optional) assumption ac may constrain the occurrence of the event. Such assumptions result in proof obligations that have to be checked when blocks get composed to modules and when modules get composed to systems.

Interfaces contain only triggered transitions. In the example, the transitions state that the history variable pending becomes true whenever a command occurs ([tr1]), and false whenever the command gets acknowledged ([tr2]). This is indicated by the prime symbol $'$. More formally, the primed occurrence of a variable represents its value in the next state. p and p' are used as shorthand instead of $p = \mathsf{true}$ and $p' = \mathsf{true}$ for Boolean valued variables.

Obviously, the desired protocol of alternating Cmd and CmdAck actions can be achieved by alternating execution of the two transitions (starting with the first). Problems arise however when the environment does not stick to the protocol. Suppose the environment does not wait for the CmdAck but sends two successive Cmd actions resulting in two successive executions of the first transition. Even worse, both the command and the corresponding acknowledgement could occur simultaneously which would result in a contradiction. Thus the question whether the protocol is followed is not local and can only be decided by the knowledge of the calling module. To avoid such global reasoning, TLT allows to note assumptions about the environment. With help of History variables this can be done in arbitrary detail and results in an abstract representation of the environment. In the example, the assumptions restrict the occurrence of the command to states where pending is false, whereas the acknowledgement may only occur whenever pending is true. These assumptions become proof obligations when the block gets composed and must be guaranteed by the blocks controlling the command and the acknowledgement. The advantage is

however, that these proof obligations can be generated and checked automatically in the sense that it is syntactically determined and independent of the concrete environment.

The Transition Relation

To extract the transition relation, first consider transition [tr1]. The informal meaning "if a command occurs, then the history variable pending is set" gets translated to the first order predicate:

(1a) $\mathsf{Cmd} \Rightarrow \mathsf{pending}'$

That is, the occurrence of Cmd implies that the state of pending becomes true.

Due to the assumption, a call only occurs in state ¬pending:

(1b) $\mathsf{Cmd} \Rightarrow \neg\mathsf{pending}$

Together what *must* happen is formalized by

(1) $\mathsf{Cmd} \Rightarrow \neg\mathsf{pending} \wedge \mathsf{pending}'$

Similarly, for the second translation one obtains

(2a) $\mathsf{CmdAck} \Rightarrow \neg\mathsf{pending}'$

without considering the assumption and

(2) $\mathsf{CmdAck} \Rightarrow \mathsf{pending} \wedge \neg\mathsf{pending}'$

as the complete translation. Here, CmdAck abbreviates $\exists_{\mathsf{c}}\ \mathsf{CmdAck}(\mathsf{c})$, which formally may be read "CmdAck occurs and there is some value transmitted". That is, an action with value-passing is abbreviated as if it was a signal without value-passing. As the exact value that is transmitted often is irrelevant, this convention will be used throughout the thesis.

In a second step, all *possible* changes of the sole controlled variable pending are determined. Due to the fact that no other module may change the value of pending, the transition relation for pending is

(3) $\mathsf{pending} \neq \mathsf{pending}' \Rightarrow \mathsf{Cmd} \vee \mathsf{CmdAck}$

that is, if pending changes its value then this is due to the events Cmd or CmdAck.

Now, two transition relations may be formulated. The first expresses what is known for sure, i.e. without relying on the assumptions, the second one assumes these assumptions to hold:

$\delta^{-\mathsf{ac}}(\mathsf{Acknowledge}) \stackrel{\text{def}}{=} (1a) \wedge (2a) \wedge (3)$

$\delta(\mathsf{Acknowledge}) \stackrel{\text{def}}{=} (1) \wedge (2) \wedge (3)$

In many other specification languages (like UNITY), transitions are executed interleaved, and the transition relation of a program is built up of the transition relations of individual transitions, typically combined by means of a program counter. In contrast, the transition relation δ introduced above might be called data-driven. It allows for the simultaneous execution of several transitions given they do not contradict.

Temporal Logic, Traces, and Fairness

Taking into account that the initially predicate specifying the initial values of the variables is simply $\neg$pending, two temporal formulas describing the protocol are:

$$\Phi^{-ac}(Acknowledge) \stackrel{\text{def}}{=} \neg \text{ pending} \wedge \Box\ \delta^{-ac}(Acknowledge) \qquad \text{and}$$

$$\Phi(Acknowledge) \stackrel{\text{def}}{=} \neg \text{ pending} \wedge \Box\ \delta(Acknowledge)$$

where $\Box$ has to be read "always", "in every step".

Temporal formulas will be interpreted over infinite sequences of valuations, called *traces*. A valuation maps variable identifiers to values of appropriate type and actions to either the value transmitted or to $\perp$ with the intended meaning that the action does not occur. For signals, $\surd$ denotes their occurrence.

Noting alternately the values of the variables and actions according to the schema

$$(\text{value of pending}) \xrightarrow{\binom{\text{value of Cmd}}{\text{value of CmdAck}}} (\text{value of pending}) \xrightarrow{\binom{\text{value of Cmd}}{\text{value of CmdAck}}} \ldots$$

the sequence

$$(\text{false}) \xrightarrow{\binom{\surd}{\perp}} (\text{true}) \xrightarrow{\binom{\perp}{\perp}} (\text{true}) \xrightarrow{\binom{\perp}{ok}} (\text{false}) \xrightarrow{\binom{\surd}{\perp}} (\text{true}) \ldots$$

is an initial segment of a trace of Φ(Acknowledge). This trace is said to be a model of Φ(Acknowledge), and vice versa, Φ(Acknowledge) is said to hold on that trace.

Instead of interpreting a trace $\zeta_0 \xrightarrow{\alpha_0} \zeta_1 \xrightarrow{\alpha_1} \ldots$ as alternating the values ζ_i of the variables and values α_i of actions, one can understand a trace also as sequence of triples $\zeta_i \xrightarrow{\alpha_i} \zeta_{i+1}$. The i-th triple, then represents the i-th evaluation of the transition predicate δ, which will be referred to as the i-th step taken by the block. Thus, ζ_i is the valuation of the unprimed variables occurring in δ (the valuation of the variables "before" the i-th step), ζ_{i+1} the valuation of the primed variables (the values "after" the i-th step) and α_i the valuation of the actions (their values "during" the i-th step). A special role play the *stutter steps* $\zeta \xrightarrow{\tau} \zeta$, where τ is the valuation assigning $\perp$ to all actions. Stutter steps trivially are a model of the transition relation δ. As a consequence, a sequence of stutter steps $\zeta \xrightarrow{\tau} \zeta \xrightarrow{\tau} \zeta \ldots$ such that ζ satisfies the initial predicate, is a trace of Φ.

On the trace given above, the first step $(\text{false}) \xrightarrow{\binom{\surd}{\perp}} (\text{true})$ is a model of the execution of the first transition, the second one $(\text{true}) \xrightarrow{\binom{\perp}{\perp}} (\text{true})$ models a stutter step with respect to the variables and actions of this interface.

Another model of both $\Phi(Acknowledge)$ and $\Phi^{-ac}(Acknowledge)$ is

$$(\mathsf{false}) \xrightarrow{\binom{\bot}{\bot}} (\mathsf{false}) \xrightarrow{\binom{\bot}{\bot}} (\mathsf{false}) \xrightarrow{\binom{\bot}{\bot}} \ldots$$

modeling the situation that the environment never issues a Cmd action.

Because $\Phi^{-ac}(Acknowledge)$ does not require the environment to obey the assumptions, there are additional models for $\Phi^{-ac}(Acknowledge)$ that are not a model of $\Phi(Acknowledge)$ like, for example,

$$(\mathsf{false}) \xrightarrow{\binom{\surd}{\bot}} (\mathsf{true}) \xrightarrow{\binom{\surd}{\bot}} (\mathsf{true}) \xrightarrow{\binom{\bot}{ok}} (\mathsf{false}) \xrightarrow{\binom{\surd}{\bot}} (\mathsf{true}) \ldots$$

where the environment emits two commands successively without waiting for an acknowledgement.

Transition Systems

Transition systems are a special kind of automata made up of a set of states connected by a set of (labelled) edges. The transition systems used in this thesis are labelled by (the valuations of) the transition relation.

Whereas there is one sole translation of a TLT specification into its logical representation, there are several ways of defining transition systems representing the specification. One canonic form are *concrete transition systems*, characterized by their state space being built up of all possible valuations of the controlled variables. Often, the concrete state space is infinite which foils using decision procedures for verification (that in worst case make an exhaustive search through the state space). To overcome infinite state spaces, *abstract transition systems* are introduced where each abstract state represents a set of concrete states. Whereas the paths through (the canonical) concrete transition system of a specification coincide with the traces that are models of the corresponding temporal formula Φ, the paths through abstract transition systems correspond to a superset of these traces. Thus one drawback of abstract transition systems is that a property may hold for all "concrete" traces but be violated by some additional traces of the abstract representation. Such situations often can be dealt with by refining the abstract representation. A second difficulty lies in finding a "good" first abstract transition system. Such a canonical abstract representation will be introduced in Chapter 5. The basic idea is to use the control information that is available in the textual representation in form of guards, assumptions and the initial predicate. In Chapter 4, a general notion of transition system is introduced that contains both the concrete and abstract transition systems.

Both for concrete and abstract transition systems, the construction of the state space is based on the controlled variables of a block. The block Acknowledge only controls one variable pending that has two possible values. Thus the concrete transition system already is finite, consisting of two states. For readability reasons, in the graphical representation of the transition system, the states are labeled by the predicates pending and ¬pending, instead of the values "true" and "false". The initial state ¬pending is marked by an additional arrow. The arc between two states ζ_1 and ζ_2 is labeled by the set of valuations that are models of the transition relation and that agree on the controlled variables with ζ_1 and ζ_2.[3] Again, these sets are represented by first

[3] Adding information about the states ζ_1 and ζ_2 to the labels simplifies the description of the composition of blocks.

order logic predicates: A label p represents the set of all valuations β such that β is a model of p. Thus, the formula used as label for a transition between two states labeled by s_1 and s_2 is given by $s_1 \wedge \delta(\mathsf{Acknowledge}) \wedge s_2'$ or any equivalent formula that has the same set of valuations as its models. Arcs labeled by the empty set (false) are omitted.

The concrete transition system representing the protocol is visualized in Figure 2.5.

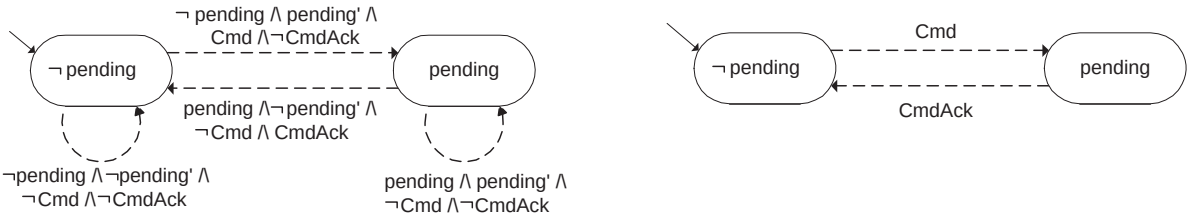


Figure 2.5: The transition system shown on the left describes the protocol with "exact" labels. On the right, the labels on the arcs are incomplete: every transition label should also state the relation with respect to the controlled variables and that (regarding the actions) nothing else happens. The stutter loops are omitted completely. These conventions get applied in all subsequent graphs.

The set of traces of a concrete transition system is obtained by following all paths starting in an initial state. As in the logical representation, the transition system corresponding to $\Phi^{-ac}(\mathit{Acknowledge})$ (see Figure 2.6) allows for additional traces.

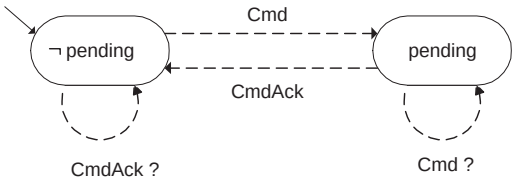


Figure 2.6: If the assumptions are not required to hold then Cmd actions may also occur in state pending, and CmdAck actions may also occur in state ¬pending. This is expressed by the "?"-notation: for an action A, A(c)? abbreviates $\mathsf{A(c)} \vee \neg\mathsf{A}$. An equivalent "exact" label for the self loop of state pending would be $\mathsf{pending} \wedge \mathsf{pending}' \wedge \neg\mathsf{CmdAck}$.

2.3.2 Modules

A module is a set of blocks that belong together, in the sense that they know the same variables and actions. More formally, they share one common set of declarations. To the environment (consisting of other modules), only a part of the declared variables is visible. The other variables are "hidden".

The Motor Module

In the textual representation, the composition to a module is achieved by the concatenation of the blocks. By convention, the "main" block is not named explicitly (nor does it start with

the keyword Block). The set of declarations that the blocks share is given as the union of the declarations that occur in the blocks.

The motor module uses the interface Acknowledge twice for a Start and a Stop command. These commands are turned into appropriate values of the Boolean valued variable motor that is connected to the real physical motor: If motor is set, then the motor is caused to start. The velocity of the motor is used to determine whether a command was successful.

Module Motor

Declarations

Variables

Read	v	: Real
Write	motor	: Boolean Init false
History	pendingStart, pendingStop	: Boolean

Actions

In	Start, Stop	: ()
Out	StartAck, StopAck	: { ok, ko }

Include Interface Acknowledge [Cmd ← Start, CmdAck ← StartAck, pending ← pendingStart]

Include Interface Acknowledge [Cmd ← Stop, CmdAck ← StopAck, pending ← pendingStop]

Transitions

[gt1] pendingStop ⟶ ¬motor′ ‖ If pendingStart Then StartAck(ko) ‖ If v=0 Then StopAck(ok)

[gt2] pendingStart ∧ ¬pendingStop ⟶ motor′ ‖ If v≠0 Then StartAck(ok)

End.

Besides the two interfaces, the module only has got one further (main) block. The transition section of this block consists of two guarded transitions [gt1] and [gt2]. [gt1] *may* be executed only if a Stop action is pending. The ‖-operator separates (sub)commands that have to be executed "in parallel". In order to syntactically check for inconsistencies, it is required that these subcommands affect disjoint sets of actions and of primed variables. If executed, [gt1] thus

(1) resets the variable motor to turn off the motor,
(2) emits a StartAck(ko) action given a pending Start command, and
(3) acknowledges the pending Stop command given the velocity of the motor equals 0.

Because only the occurrence of StopAck(ok) can reset pendingStop and thereby falsify the guard, the transitions gets executed until the physical motor is stopped. Similarly, [gt2] *may* be executed only if a Start action but no Stop action is pending. Its effect is to set the variable motor and to acknowledge the Start command as soon as the motor is running.

The Transition Relation of Module motor

The transition relation of module motor is built up from the transition relations of the three blocks of the module. Firstly, the transition relation of the main block is determined:

$$\begin{aligned}\delta(\mathsf{main}) \stackrel{\text{def}}{=} &(\text{motor}\neq\text{motor}' \Rightarrow (\text{gt1}) \vee (\text{gt2})) \wedge \\ &(\text{StartAck} \Rightarrow (\text{gt1}) \vee (\text{gt2})) \wedge \\ &(\text{StopAck} \Rightarrow (\text{gt1}))\end{aligned}$$

where

$$\begin{aligned}(\text{gt1}) \stackrel{\text{def}}{=} &\text{pendingStop} \wedge \neg\text{motor}' \wedge \\ &(\text{pendingStart} \wedge \text{StartAck(ok)} \vee \neg\text{pendingStart} \wedge \neg\text{StartAck}) \wedge \\ &(\text{v}=0 \wedge \text{StopAck(ok)} \vee \text{v}\neq 0 \wedge \neg\text{StopAck}) \wedge\end{aligned}$$

$$\begin{aligned}(\text{gt2}) \stackrel{\text{def}}{=} &\text{pendingStart} \wedge \neg\ \text{pendingStop} \wedge \text{motor}' \wedge \\ &(\text{v}\neq 0 \wedge \text{StartAck(ok)} \vee \text{v}=0 \wedge \neg\text{StartAck})\end{aligned}$$

Note, that guarded transitions $[\!] \ \mathsf{g} \longrightarrow \mathsf{cmd}$ enter the transition relation as conjunction $\mathsf{g} \wedge \text{Trl}(\mathsf{cmd})$. Here, Trl is a function that translates a command into a first order logic formula. In the example, the $\|$-operator for simultaneous execution (in the sense of within one atomic step) is replaced by the logical "and", whereas the If b Then cmd-construct gets translated to $\mathsf{b} \wedge \text{Trl}(\mathsf{cmd}) \vee \neg\mathsf{b} \wedge \mathsf{Stutter}(\text{Ctr}(\mathsf{cmd}))$. That is, if the condition b holds, then the translation of the command is in effect. If the condition b does not hold, then the variables and actions controlled by cmd stutter. For variables, stuttering means, that they do not change their value. For actions, stuttering means that they do not take place.

Based on the transition relation, the provisional temporal formula

$$\Delta(\mathsf{main}) \stackrel{\text{def}}{=} \Box\, \delta(\mathsf{main})$$

describes the so-called safety part of the specification. It remains to determine some scheduling for the guarded transitions. Without such a scheduling, one possible trace of $\Delta(\mathsf{main})$ consists of stuttering forever, even in case of some pending command. However, in general, one wants to verify properties that hold for all traces. Allowing this trace thus would mean that the essential liveness property of the protocol (namely that command eventually get acknowledged) could not be verified.

At first glance, one possibility to exclude this trace lies in disallowing stuttering. In the definition of δ stuttering was explicitly allowed for each controlled variable and action[4]. The possible stuttering gets obvious by reformulating $\delta(\mathsf{main})$ as

$$\begin{aligned}\delta(\mathsf{main}) \stackrel{\text{def}}{=} &(\text{motor}=\text{motor}' \vee (\text{gt1}) \vee (\text{gt2})) \wedge \\ &(\neg\text{StartAck} \vee (\text{gt1}) \vee (\text{gt2})) \wedge \\ &(\neg\text{StopAck} \vee (\text{gt1}))\end{aligned}$$

Obviously, this formula is fulfilled if nothing happens. Yet, stuttering is essential for a general notion of composition that does not force all blocks to do a "real" non stutter transition in each step (besides, stuttering is even more crucial for refinement purposes). Therefore, one can not simply forbid stuttering.

Fortunately, there is a less strict requirement that excludes only infinite stuttering: one forbids that the transitions continuously are enabled but never get executed. This can be reformulated

[4]Non controlled variables and actions are not restricted at all by δ.

positively as the property "if continuously *some* guard is enabled then eventually the block gets a chance to execute (at least) *one* of its guarded transitions". This property is called *local progress* and assumed for all blocks. Therefore, it is not noted explicitly in the textual representation.

Local progress is a special case of a (weak) fairness condition. Although such fairness conditions in general will reduce the number of traces, it can be guaranteed under reasonable preconditions that they can not forbid all traces. Even better, one can prove this property in a constructive manner resulting in a simple scheduler that only produces traces that are weak fair.

The temporal formula that corresponds to the textual specification can now be completed:

$$\Phi(main) \stackrel{\text{def}}{=} \Delta(main) \wedge \mathcal{WF}(\text{ someGuard, } (\text{gt1}) \vee (\text{gt2}) \text{ })$$

where

$$\text{someGuard} \stackrel{\text{def}}{=} \text{pendingStop} \vee \text{pendingStart} \wedge \neg\text{pendingStop}$$

Actually, $\mathcal{WF}(\mathsf{g}, \mathsf{gt})$ is a shorthand for the temporal logic formula $\Diamond\Box\mathsf{g} \Rightarrow \Box\Diamond\,\mathsf{gt}$. $\Diamond$ may be read "eventually" and thus $\Diamond\Box$ means "from some point continuously" and $\Box\Diamond$ can be interpreted as "always eventually" or as "infinitely often". Putting these interpretations together, $\mathcal{WF}(\mathsf{g}, \mathsf{gt})$ is said to hold on a trace, if it is true that if (on that trace) g holds from some point continuously, then gt holds infinitely often.

The temporal logic also serves as property language in TLT. For example, the formula $\Delta(\mathsf{main}) \Rightarrow \Box\neg\mathsf{StopAck(ko)}$ expresses that the main block never returns that the Stop command has failed. For such "safety"-properties, it is always sufficient to consider $\Delta(\mathsf{main})$ instead of $\Phi(\mathsf{main})$ which justifies the introduction of $\Delta(\mathsf{main})$.

The transition system corresponding to the main block is shown in Figure 2.7. In the figure, the transitions are drawn by solid lines to indicate that the transitions are influenced by some fairness condition. In the example, a further distinction is not necessary, because there is only one fairness condition. Transitions not restricted by any fairness condition are drawn by dashed lines as in Figure 2.5.

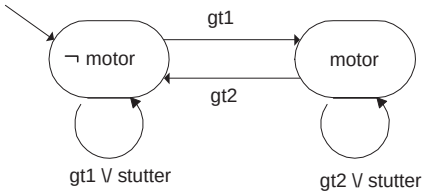


Figure 2.7: Because the only controlled variable motor is Boolean-valued, the concrete transition system for the main block consists of only two states.

Again, the set of traces of the transition system is obtained by following all paths starting in an initial state and obeying the fairness conditions.

Composition of Blocks, Assumption Commitment Reasoning

Combining blocks is essential for describing large systems. Therefore, it is crucial to have a straightforward notion of composition guaranteeing that

(1) a block has the same meaning in all contexts (its meaning is *fully abstract*),
(2) blocks may not "contradict" each other, and
(3) the meaning of the composition of blocks should be simple (describable as "and").

The first item has already be dealt with by defining a unique set of traces as the semantics of a block. The second item will be dealt with in detail in Chapter 4 and in Chapter 5. In essence, blocks can not contradict because they restrict the behavior of different sets of "controlled" variables and actions. For the last item see Figure 2.8: composing two blocks means

(1) syntactical concatenation in the textual,
(2) conjunction of the temporal formulas in the logical,
(3) forming product automata in the transition system, and
(4) intersection in the trace

representation.

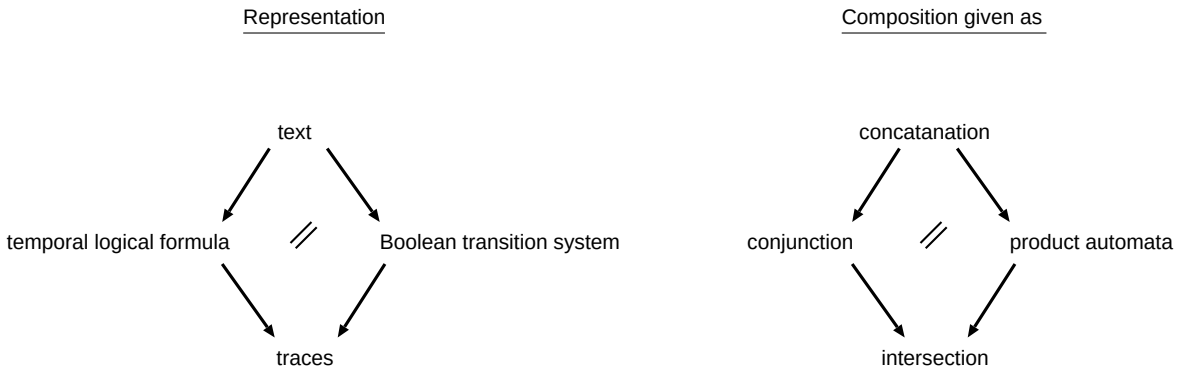


Figure 2.8: Composition in the different views of a specification.

For blocks to be composed to one module, a series of consistency criteria have to be fulfilled in order to guarantee that the blocks may not "contradict" each other. The main property blocks have to fulfill is that they "control" different variables and actions. This means, for example, that only one block may issue a StartAck action or change (write) the variable motor.

Besides, before actually composing the blocks, the assumptions of the interfaces have to be checked. The assumptions regarding the commands have to be committed by the environment that controls the Start and Stop action and thus does not constrain the composition of the three blocks forming the motor module. However, the main block controls the two acknowledgements. Thus, the module has to guarantee that the corresponding command is pending whenever the main block emits an acknowledgement. In case of the motor module this is trivial because the transition relation of the main block already implies the assumptions:

(CC1) $\delta(main) \Rightarrow (StopAck \Rightarrow pendingStop)$

(CC2) $\delta(main) \Rightarrow (StartAck \Rightarrow pendingStart)$

Therefore, no temporal reasoning is required.[5] The consistency condition (CC1) follows immediately from the fact that the StopAck action is guarded by pendingStop in transition [gt1] and

[5] One is tempted to say, the reasoning is purely syntactical.

does not occur elsewhere. StartAck occurs in both transitions: In [gt1] the condition of the If-command equals pendingStart, in [gt2] the guard implies pendingStart and thus (CC2) holds too.

As a result, the three blocks may be composed to form a module. Let δ(I1) and δ(I2) denote the transition relations of the interfaces, that is,

$$\begin{aligned}
\delta(\mathrm{I1}) \stackrel{\mathrm{def}}{=} & (\ \mathrm{Start} \ \Rightarrow \neg\mathrm{pendingStart} \wedge \mathrm{pendingStart}') \wedge \\
& (\ \mathrm{StartAck} \ \Rightarrow \mathrm{pendingStart} \wedge \neg\mathrm{pendingStart}') \wedge \\
& (\ \mathrm{pendingStart} \neq \mathrm{pendingStart}' \Rightarrow \mathrm{Start} \vee \mathrm{StartAck}\ )
\end{aligned}$$

$$\begin{aligned}
\delta(\mathrm{I2}) \stackrel{\mathrm{def}}{=} & (\ \mathrm{Stop} \ \Rightarrow \neg\mathrm{pendingStop} \wedge \mathrm{pendingStop}') \wedge \\
& (\ \mathrm{StopAck} \ \Rightarrow \mathrm{pendingStop} \wedge \neg\mathrm{pendingStop}') \wedge \\
& (\ \mathrm{pendingStop} \neq \mathrm{pendingStop}' \Rightarrow \mathrm{Stop} \vee \mathrm{StopAck}\ )
\end{aligned}$$

Then the transition relation for the module is defined as the conjunction of the transition relations of the three blocks:

$$\begin{aligned}
\delta(\mathsf{Motor}) \stackrel{\mathrm{def}}{=} & \ \delta(\mathrm{main}) \wedge \delta(\mathrm{I1}) \wedge \delta(\mathrm{I2}) \\
\equiv & \ \neg\mathrm{pendingStart} \wedge \neg\mathrm{pendingStop} \wedge \mathrm{motor}'{=}\mathrm{motor} \wedge \\
& \quad (\ \mathrm{Start} \wedge \mathrm{pendingStart}' \vee \neg\mathrm{Start} \wedge \neg\mathrm{pendingStart}') \wedge \\
& \quad (\ \mathrm{Stop} \wedge \mathrm{pendingStop}' \vee \neg\mathrm{Stop} \wedge \neg\mathrm{pendingStop}') \wedge \\
& \quad \neg\mathrm{StartAck} \wedge \neg\mathrm{StopAck} \\
& \vee \neg\mathrm{pendingStart} \wedge \mathrm{pendingStop} \wedge \mathrm{motor}'{=}\mathrm{motor} \wedge \\
& \quad \mathrm{Start} \wedge \neg\mathrm{Stop} \wedge \neg\mathrm{StartAck} \wedge \neg\mathrm{StopAck} \wedge \mathrm{pendingStart}' \wedge \mathrm{pendingStop}' \\
& \vee \neg\mathrm{pendingStart} \wedge \mathrm{pendingStop} \wedge \neg\mathrm{motor}' \wedge \\
& \quad (\ \mathrm{Start} \wedge \mathrm{pendingStart}' \vee \neg\mathrm{Start} \wedge \neg\mathrm{pendingStart}') \wedge \\
& \quad \neg\mathrm{Stop} \wedge \neg\mathrm{StartAck} \wedge \\
& \quad (\ \mathrm{v}{=}0 \wedge \mathrm{StopAck(ok)} \wedge \mathrm{pendingStop}' \vee \mathrm{v}{\neq}0 \wedge \neg\mathrm{StopAck} \wedge \neg\mathrm{pendingStop}') \\
& \vee \mathrm{pendingStart} \wedge \neg\mathrm{pendingStop} \wedge \mathrm{motor}'{=}\mathrm{motor} \wedge \\
& \quad \neg\mathrm{Start} \wedge \mathrm{Stop} \wedge \neg\mathrm{StartAck} \wedge \neg\mathrm{StopAck} \wedge \mathrm{pendingStart}' \wedge \mathrm{pendingStop}' \\
& \vee \mathrm{pendingStart} \wedge \neg\mathrm{pendingStop} \wedge \mathrm{motor}' \wedge \\
& \quad (\ \mathrm{Stop} \wedge \mathrm{pendingStop}' \vee \neg\mathrm{Stop} \wedge \neg\mathrm{pendingStop}') \wedge \\
& \quad \neg\mathrm{Start} \wedge \neg\mathrm{StopAck} \wedge \\
& \quad (\ \mathrm{v}{\neq}0 \wedge \mathrm{StartAck(ok)} \wedge \mathrm{pendingStart}' \vee \mathrm{v}{=}0 \wedge \neg\mathrm{StartAck} \wedge \neg\mathrm{pendingStart}') \\
& \vee \mathrm{pendingStart} \wedge \mathrm{pendingStop} \wedge \neg\mathrm{motor}' \wedge \neg\mathrm{Start} \wedge \neg\mathrm{Stop} \wedge \mathrm{StartAck(ko)} \wedge \\
& \quad (\ \mathrm{v}{=}0 \wedge \mathrm{StopAck(ok)} \wedge \mathrm{pendingStop}' \vee \mathrm{v}{\neq}0 \wedge \neg\mathrm{StopAck} \wedge \neg\mathrm{pendingStop}')
\end{aligned}$$

The treatment of assumptions differs in two ways (quite significantly) from the one found elsewhere in the literature (e.g. in [CK93],[Col94],[AL93],[DF95], or [Mer94]). Firstly, the reasoning about assumptions is lifted to the textual representation of specifications where (safety) assumptions occur as annotations. In other works assumptions are considered to be a property of specifications expressed either in logical or semantical terms. Secondly, as seen above, assumptions often are structural properties of specifications. In these cases it is not necessary in TLT to use temporal logic for the proofs. In other approaches, assumption commitment style is always carried out using terms of temporal reasoning. TLA, for example, uses expressions of the form $A \xrightarrow{+} C$ with the intended meaning "if the environment respects the assumptions A up to point i, then the module commits C up to point i+1".

Liveness assumptions are dealt with on the level of layered system descriptions. That means

that the underlying "architecture" of the system can be used to show that cyclic reasoning is avoided.

Modules in Temporal Logic

The composition of blocks to modules in the temporal logic view is achieved by conjuncting the temporal formulas representing the blocks and then hiding the local and the history variables:

$$\Phi(Motor)^{vis} \stackrel{\text{def}}{=} \exists_{\mathsf{pending}} \ (\Phi(main) \wedge \Phi(I1) \wedge \Phi(I2) \ )$$

Here, $\exists_{\mathsf{pending}}$ means that there is a sequence of values for the variable pending such that $\Phi(main)$ $\wedge \Phi(I1) \wedge \Phi(I2)$ holds. As a consequence of hiding, the valuations for pending are not mentioned any more in the traces of $\Phi(Motor)$.

The history variable pending is is an auxiliary variable in the sense of [AL88]. This means that it is completely determined by Cmd and Return.

Initially, it is constant: ¬pending. Then, in each step, it is given as a function of Cmd and CmdAck according to the following function table:

value of Cmd	value of CmdAck	value of pending′
$\perp$	$\perp$	value of pending
$\surd$	$\perp$	true
$\perp$	$\in \{ok, ko\}$	false
$\surd$	$\in \{ok, ko\}$	forbidden

The situation where both Cmd and CmdAck occur may not arise, because of the mutual exclusive assumptions of [tr1] and [tr2].

Modules as Transition Systems

A concrete transition system of module Motor is shown in Figure 2.9. Its state space is obtained by forming all combinations of the states of the interfaces with the states of block main. As the controlled variables are disjoint, all combinations (interpreted as conjunctions) are satisfiable by construction. Two states of the composition are connected by an arc labeled by a predicate equivalent to $\exists_{\mathsf{pending},\mathsf{pending}'}(\lambda(\mathsf{I1}) \wedge \lambda(\mathsf{I2}) \wedge \lambda(\mathsf{main}))$ if and only if the corresponding states in the components are connected by labels λ(I1), λ(I2) and λ(main) respectively, and their conjunction is satisfiable.

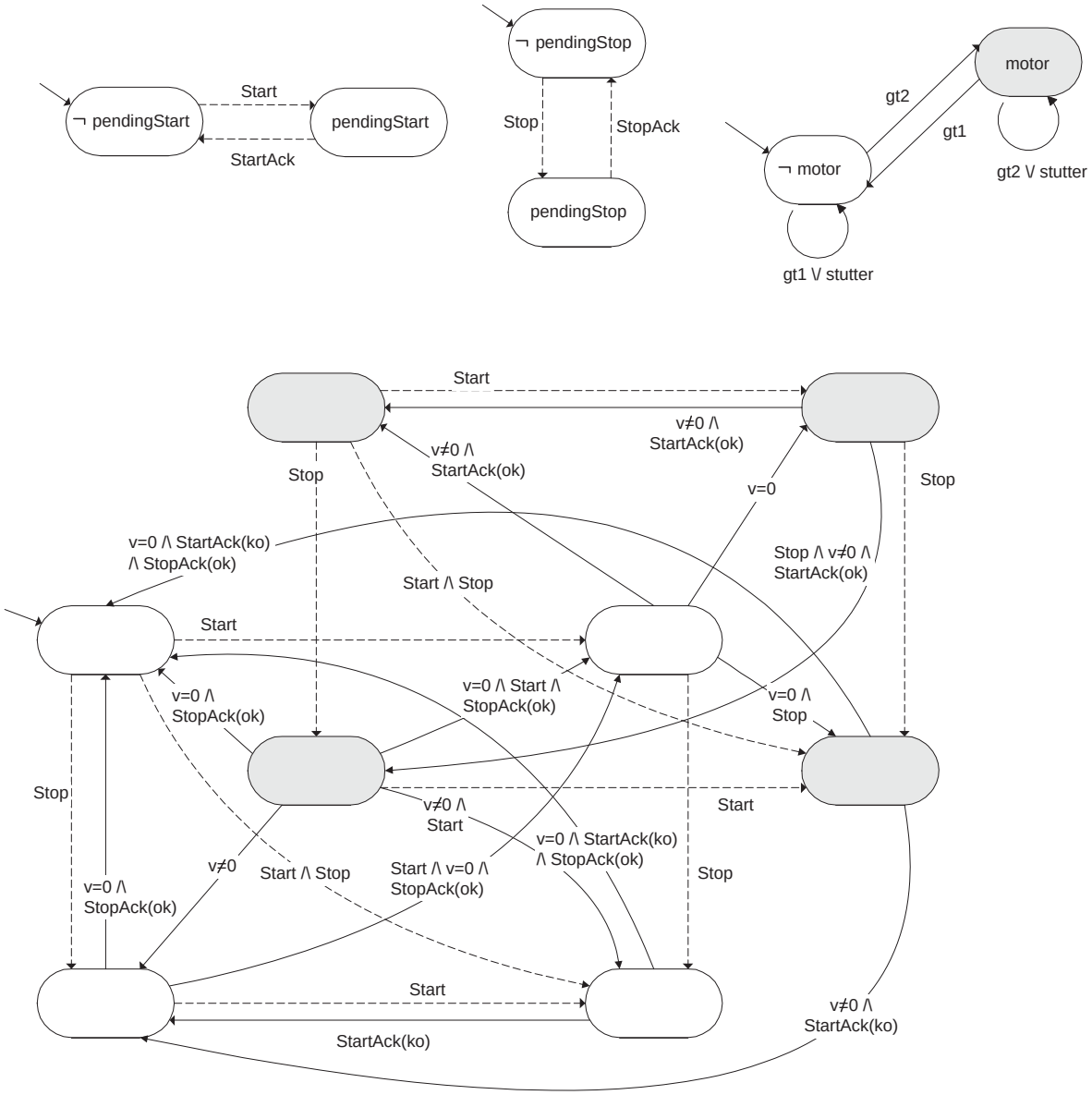


Figure 2.9: On top, the transition systems of the three blocks are repeated. On bottom, their product is shown. For clarity, all states where the variable motor is set are grayed.

The Other Modules

The two remaining modules do not introduce new theoretical concepts and are therefore presented only in their textual representations.

The Module StopCondition reads a variable error that signals an exception in the process that makes it necessary to halt the group. Besides, the operator may want to stop the motors using the input action OS_StopGroup. As a consequence, the transition section contains both a guarded and two triggered transitions. The former one forwards the exception in the process by turning the variable error into an action StopGroup, the latter ones deal with the incoming actions.

The specification variable result used in the last triggered transition serves as a formal parameter binding the value of the StopGroupAck action.

The AutoStop action finally informs the operator that the motors had to be stopped due to some exception in the process. In a more realistic example, there would be more than one cause of error, and as a consequence the AutoStop action would also return some alarm text to the operator station.

Module StopCondition

Declarations

Variables

Read	error	: Boolean
History	pndOSStopGroup	: Boolean
History	pndStopGroup	: Boolean
Spec	result	: { ok, ko }

Actions

In	OS_StopGroup	: ()
Out	OS_StopGroupAck	: { ok, ko }
Out	AutoStop	: ()
Out	StopGroup	: ()
In	StopGroupAck	: { ok, ko }

Include Interface Acknowledge [Cmd ← OS_StopGroup, CmdAck ← OS_StopGroupAck, pending ← pndOSStopGroup]

Include Inverted Interface Acknowledge [Cmd ← StopGroup, CmdAck ← StopGroupAck, pending ← pndStopGroup]

Transitions

$[\!]$ OS_StopGroup $\Rightarrow\!\triangleright$ StopGroup

$[\!]$ error $\wedge$ $\neg$ pndStopGroup $\longrightarrow$ StopGroup

$[\!]_{\text{result}}$ StopGroupAck(result) $\Rightarrow\!\triangleright$ If pndOSStopGroup Then OS_StopGroupAck(result)
Else AutoStop

End.

Module GroupCtr

Parameters

nbMotors : Integer

Declarations

Variables

History pendingStartGroup, pendingStopGroup : Boolean
History pendingStart, pendingStop : Vector [1..nbMotors] Of Boolean
Local motor : Vector [1..nbMotors] Of { stopped, pndStart, pndStop, running }

Actions

In StartGroup, StopGroup : ()
Out StartGroupAck, StopGroupAck : { ok, ko }
Out Start, Stop : Vector [1..nbMotors] Of ()
In StartAck, StopAck : Vector [1..nbMotors] Of { ok, ko }

Include InterfaceAcknowledge [Cmd ← StartGroup, CmdAck ← StartGroupAck, pending ← pendingStartGroup]

Include InterfaceAcknowledge [Cmd ← StopGroup, CmdAck ← StopAckGroup, pending ← pendingStopGroup]

Include ⟨ m : [1..nbMotors] ⟩ Inverted Interface Acknowledge [Cmd ← Start[m], CmdAck ← StartAck[m], pending ← pendingStart[m]]

Include ⟨ m : [1..nbMotors] ⟩ Inverted Interface Acknowledge [Cmd ← Stop[m], CmdAck ← StopAck[m], pending ← pendingStop[m]]

Transitions

⟨ m : [1..nbMotors] ⟩[t1] StartAck[m](ok) =▷ motor[m]′= running

⟨ m : [1..nbMotors] ⟩[t2] StartAck[m](ko) ∨ StopAck[m] =▷ motor[m]′= stopped

⟨ m : [1..nbMotors] ⟩[t3] pendingStartGroup ∧ ¬pendingStopGroup ∧
$\bigwedge_{\langle\, n : [1..nbMotors] \,\rangle}$ (n<m ⇒ motor[n] = running) ∧ motor[m] = stopped
⟶ Start[m] ‖ motor[m]′= pndStart

[t4] pendingStartGroup ∧ ¬pendingStopGroup ∧
$\bigwedge_{\langle\, m : [1..nbMotors] \,\rangle}$ (motor[n] = running)
⟶ StartGroupAck(ok)

⟨ m : [1..nbMotors] ⟩[t5] pendingStopGroup ∧ motor[m] ∈ { running, pndStart } ∧
$\bigwedge_{\langle\, n : [1..nbMotors] \,\rangle}$ (n>m ⇒ motor[n] = stopped)
⟶ Stop[m] ‖ motor[m]′= pndStop

[t6] pendingStopGroup ∧ $\bigwedge_{\langle\, m : [1..nbMotors] \,\rangle}$ (motor[n] = stopped)
⟶ StopGroupAck(ok) ‖
If pendingStartGroup Then StartGroupAck(ko)

End.

The GroupCtr module is responsible for successively starting and stopping the motors. In order to deal with an arbitrary number of motors, both variables, actions, transitions and even interfaces are parameterized. As an example, consider the local variable motor. Actually, as is the case for all other parameterizations, the parameterized declaration for motor is a schema for |nbMotors| declarations. The resulting variables are named motor[1], ... motor[nbMotors]. They are used to keep track of the states of the motors. This is necessary to maintain the order in which the motors are started and stopped. If, for example, an exception occurs in the process while the motors get started, then the last motor that is either running or that received a pending Start command is stopped first (compare transition [t5]).

The parameterization for the transitions [t1] is a schema for the following |nbMotors| transitions:

$$
\begin{array}{lll}
[\text{t1\_1}] & \text{StartAck}[1](\text{ok}) \;\; =\!\triangleright & \text{motor}[1]' = \text{running} \\
\vdots & & \\
[\text{t1\_nbMotors}] & \text{StartAck}[\text{nbMotors}](\text{ok}) \;\; =\!\triangleright & \text{motor}[\text{nbMotors}]' = \text{running}
\end{array}
$$

That is, the occurrence of StartAck[i](ok) triggers motor[i] to be set to state running.

Similarly, each of the two parameterized interfaces abbreviates |nbMotors| interfaces. In case of the interfaces, one has to pay attention to first do the substitutions and then duplicate the interface. In order to match the corresponding interfaces of the motor modules, the parameterized interfaces have to be inverted.

2.3.3 Systems

A system is built up of instances of modules that optional may be ordered in layers. Each layer then relies on a set of properties of lower layers and guarantees a set of properties to upper layers. The properties on which the lowest layer (and thus the whole system) relies upon are called system assumptions. The system assumptions must be fulfilled by the environment of the system. Sometimes, it is useful to consider several sets of system assumptions for the environment in order to evaluate the effect on the properties of the system. Furthermore, it is worth noting that any effort spent on specifying system assumptions and properties is optional. Several important properties like freedom of deadlock directly result from proof obligations that are extracted automatically from the textual notation and therefore do not need to be denoted explicitly. However, the extra effort usually pays off in reduced testing time and should be standard at least for safety-critical systems or systems that are delivered in big quantities.

The following system description completes the one presented earlier in the engineering section. In the example, the system assumptions close the system with respect to the process and the operator station that are not modeled explicitly. They form a kind of minimal substitutes for the missing environment and model, for example, that the environment sticks to the acknowledge-protocol. The last two system assumptions state that the motor eventually react to the motor variable as expected: if for one of the motors the variable motor is continuously set unless the velocity indicates that the motor is running, then eventually it will be running and, vice versa, if for one of the motors the variable motor is continuously reset unless the velocity indicates that the motor is stopped, then eventually it will be stopped. These latter two assumptions are necessary to prove the essential liveness properties of the system.

The first property is concerned about starting the motors: if the group receives a StartGroup command then eventually a state will be reached where the StartGroup command is overridden by a stop condition or where all the motors are running. The second property states that both an exception in the process and a StopGroup command lead to a state where all the motors are stopped.

System MotorGroup

Parameters

MotorNo : Natural := 2

Properties

$\|\!\!\|$ □ (StartGroup ⇒ ◇ (StopGroup ∨ error ∨ $\bigwedge_{m:[1..MotorNo]}$ v[m] ≠ 0))

$\|\!\!\|$ □ (error ∨ StopGroup ⇒ ◇ $\bigwedge_{m:[1..MotorNo]}$ v[m] = 0))

Layer 1

Include Module StopCondition

Include Module GroupCtr [nbMotors ← MotorNo]

Include ⟨ m : [1..MotorNo] ⟩ Module Motor [Start ← Start[m], StartAck ← StartAck[m],
Stop ← Stop[m], StopAck ← StopAck[m],
v ← v[m], motor ← motor[m]]

Systemassumptions

$\|\!\!\|$ □ (StartGroup ⇒ ¬pendingStartGroup)

$\|\!\!\|$ □ (OS_StopGroup ⇒ ¬pndOSStopGroup)

$\|\!\!\|_{m:[1..MotorNo]}$ (motor[m] Unless v[m] ≠ 0) ⇒ ◇ (v[m] ≠ 0 ∨ error)

$\|\!\!\|_{m:[1..MotorNo]}$ (¬motor[m] Unless v[m] = 0) ⇒ ◇ v[m] = 0

End

Chapter 3

Logic

In this chapter, a logic is introduced that provides a means both for describing and for reasoning about distributed systems. As specifications of distributed and embedded systems typically have some ongoing behavior, the logic of choice is a temporal logic that allows reasoning about systems that change in time. Nevertheless, following the approach of TLA (see [Lam94a]), the reasoning about specifications in the logical view will be done in first-order logic whenever possible. This will be realized by specifying the next-state relation in 'pure' first-order logic by using primed variables to specify the next-state values. This has the advantage that a large part of the proof burden can be carried out using a 'pure' first-order calculus. In order to provide a means to describe synchronous communication, the standard first-order logic is extended to include sorted actions.

As a consequence, this chapter is divided into two sections. The first section briefly introduces the extended first-order predicate logic that allows to describe transitions of distributed systems. In the second section, this first-order logic is embedded into a temporal logic in order to deal with sequences of transitions.

3.1 A First-order Logic for Transitions

In computer science it is common to use first-order predicate logic to formally describe assertions that hold in a certain state of a computation, represented by the values of a set of variables in some computer memory. The foundation to this use of predicate logic were laid by Robert Floyd and C.A.R. Hoare in the late sixties (see [Flo67] and [Hoa69]). For example, the assertion $\mathsf{counter} = \mathsf{0}$ states that a variable $\mathsf{counter}$ has the value $\mathsf{0}$. Floyd labeled the edges of flaw charts with such assertions with the intended meaning that the assertions hold every time execution reaches the edge. Hoare uses assertions P and Q to describe the state of a computation before and after the execution of a sequence of commands S, resulting in Hoare triples $\{\mathsf{P}\}\,\mathsf{S}\,\{\mathsf{Q}\}$ (for an overview see, for example, [Gri81]).

In this thesis, first-order logic is not used solely to express assertions about the state of a computation, but rather to directly describe transitions occurring in computations. More precisely, the complete next-state relation of a specification is expressed by a first-order predicate logic formula.

This section does not provide a comprehensive introduction to logics but assumes some familiarity with first-order logic as may be gained from [Fit90], [Sch89], [Ric89] or [And86].

3.1.1 Syntax of a First-order Logic

The definition of a formal language starts by fixing a signature, sometimes also called the alphabet of the language. As usual, this signature comprises symbols for variables, functions and predicates. To reflect the data types used in the programming notation of TLT, the logic is typed and thus its signature also provides a set of symbols for sorts. Besides, a set of action symbols is included. Actions will be used for synchronization and communication purposes. If actions are used for synchronization only and do not carry a value to be communicated then they are called signals. In this special role they are said to be of "unit" sort.

Definition 3.1.1 (Signature)
Assume a set of symbols, containing disjoint non-empty sets Γ_{Var} (for variables), Γ_{Act} (for actions), Γ_{Func} (for functions), and Γ_{Pred} (for predicates).The symbols of $\Gamma_{Var} \cup \Gamma_{Act} \cup \Gamma_{Func}$ are sorted using symbols from Γ_{Sort} (for sorts). To define Γ_{Sort}, some set of elementary sort identifiers is assumed , containing at least one sort $()$ called the "unit" sort. Γ_{Sort} is assumed to be closed under forming tuples. That is, if $S_1, S_2 \in \Gamma_{Sort}$ then also the Cartesian product $S_1 \times S_2 \in \Gamma_{Sort}$[1].

The set of variables $\Gamma_{Var} = \Gamma_V \dot\cup \Gamma_{V'} \dot\cup \Gamma_S$ consists of matching sets of unprimed $\Gamma_V = \{x, y, z, \ldots, x[1], x[2], \ldots\}$ and primed $\Gamma_{V'} = \{x', y', z', \ldots, x[1]', x[2]', \ldots\}$ variables, together with a set $\Gamma_S = \{i, j, k, \ldots\}$ of specification variables. The variables $x \in \Gamma_V$ are called program variables or flexible variables.

Furthermore, there are two functions sort *and* arity. *The function* sort *maps each symbol $s \in \Gamma_{Var} \cup \Gamma_{Act} \cup \Gamma_{Func}$ to its sort $\mathrm{sort}(s) \in \Gamma_{Sort}$. Any matching pair of primed and unprimed variable is required to be of the same sort, that is, $\mathrm{sort}(x) = \mathrm{sort}(x')$ for all $x \in \Gamma_V$. The function* arity *maps each symbol $s \in \Gamma_{Pred} \cup \Gamma_{Func}$ to a tuple of sorts, written $\mathrm{arity}(s) = S_1 \times \cdots \times S_n \in (\Gamma_{Sort})^n$, such that $n \geq 1$.* sort *and* arity *are extended as described below to terms.*

Based on the signature defined above, terms and formulas are defined inductively, together with two sets FV and FAct for free variables and free actions occurring in them.

Definition 3.1.2 (Terms)
The set of terms over Γ_{Var} and Γ_{Func}, and the set $\mathsf{FV}(t)$ of free variables of a term t, are defined as:

1. *Every variable $v \in \Gamma_{Var}$ with $\mathrm{sort}(v) = S \in \Gamma_{Sort}$ is a term of sort S.*

   $\mathsf{FV}(v) = \{v\}$

2. *Given terms $t_1, \ldots, t_n$ of $\mathrm{sort}(t_i) = S_i$ and a function $f \in \Gamma_{Func}$ of $\mathrm{arity}(f) = S_1 \times \cdots \times S_n$ and $\mathrm{sort}(f) = S$, then $f(t_1, \ldots, t_n)$ is a term of sort S.*
   As a special case, unary functions with arity $()$ are called constants and denoted by f instead of $f(t_1)$.

[1] For sets A and B, their Cartesian product is made up from all pairs of elements of A and B, that is, $A \times B \stackrel{\mathrm{def}}{=} \{(a, b) \mid a \in A, b \in B\}$.

$$\mathsf{FV}(f(t_1, \ldots, t_n)) = \bigcup_{i:1,\ldots,n} \mathsf{FV}(t_i)$$

3. *Given terms t_1, t_2 of* $\mathrm{sort}(t_i) = S_i$, *then (t_1, t_2) is a term of sort $S_1 \times S_2$.*

   $$\mathsf{FV}((t_1, t_2)) = \mathsf{FV}(t_1) \cup \mathsf{FV}(t_2)$$

A term is called closed if no variables occur in it. The set FAct *is empty for all terms.*

It is important to note that actions do not occur in terms. Actions will be restricted to the role of atomic formulas defined below, and the syntax $\mathsf{A}(\mathsf{5})$ is used to denote intuitively something like "the value sent on channel A is $\mathsf{5}$" or "a synchronization on action A takes place and a value of $\mathsf{5}$ is passed". If, alternatively, actions were used syntactically like variables, then terms like $\mathsf{A} + \mathsf{5}$ without intuitive semantics could be formed. Furthermore, this avoids any confusion with variables. As an important special case, actions can also have the sort (), or unit. In this case, the syntax A abbreviates $\mathsf{A}(\mathsf{t})$ and represents "there is a signal on channel A" or "a synchronization on action A takes place".

The following table summarizes the syntactic differences of variables, actions, functions and predicates:

symbol type	arity	sort	used in terms
variables		defined	yes
actions		defined	no
functions	defined	defined	yes
predicates	defined		no

Definition 3.1.3 (Atomic Formulas)
The set of atomic formulas over predicates Γ_{Pred} and actions Γ_{Act} (using terms as defined above), and the sets $\mathsf{FV}(\phi)$ and $\mathsf{FAct}(\phi)$ of free variables and actions of an atomic formula ϕ, are defined as:

1. *Given terms $t_1, \ldots, t_n$ of* $\mathrm{sort}(t_i) = S_i$ *and a predicate $p \in \Gamma_{Pred}$ of* $\mathrm{arity}(p) = S_1 \times \cdots \times S_n$, *then $p(t_1, \ldots, t_n)$ is an atomic formula.*
   As a special case, predicates of arity () are called propositional constants and denoted by p instead of $p(t_1)$.

   $$\mathsf{FV}(p(t_1, \ldots, t_n)) = \bigcup_{i:1,\ldots,n} \mathsf{FV}(t_i) \quad \textit{and} \quad \mathsf{FAct}(p(t_1, \ldots, t_n)) = \emptyset$$

2. *Given a term t and an action $A \in \Gamma_{Act}$, both of the same sort $S \in \Gamma_{Sort}$, $A(t)$ is an atomic formula.*
   As a special case, actions with sort () are called signals. For signals, $A(t)$ will be abbreviated simply by A.

   $$\mathsf{FV}(A(t)) = \mathsf{FV}(t) \quad \textit{and} \quad \mathsf{FAct}(A(t)) = \{A\}$$

3. *Given terms t_1, t_2 of sort S, then $t_1 = t_2$ is an atomic formula.*

   $$\mathsf{FV}(t_1 = t_2) = \mathsf{FV}(t_1) \cup \mathsf{FV}(t_2) \quad \textit{and} \quad \mathsf{FAct}(t_1 = t_2) = \emptyset.$$

The definition of formulas is based on atomic formulas and includes existential and universal quantification over actions. $\exists_A$ may be read as "for some value (sendable) on channel A, or

for an empty channel A", and $\forall_A$ as "for all values (sendable) on channel A, and for an empty channel A".

Definition 3.1.4 (Syntax of First-Order Logic Formulas)
The set of formulas over predicates and terms (as defined above), and the set $\mathsf{FV}(\phi)$ *of free variables of a formula* ϕ*, are defined as:*

1. *Any atomic formula* ψ *is a formula* ϕ.

   $\mathsf{FV}(\phi) = \mathsf{FV}(\psi) \quad \textit{and} \quad \mathsf{FAct}(\phi) = \mathsf{FAct}(\psi)$

2. *If* ϕ *is a formula, then* $\neg\phi$ *is a formula.*

   $\mathsf{FV}(\neg\phi) = \mathsf{FV}(\phi) \quad \textit{and} \quad \mathsf{FAct}(\neg\phi) = \mathsf{FAct}(\phi)$;

3. *If* ϕ_1, ϕ_2 *are formulas, then* $\phi_1 \wedge \phi$ *is a formula.*

   $\mathsf{FV}(\phi_1 \wedge \phi_2) = \mathsf{FV}(\phi_1) \cup \mathsf{FV}(\phi_2) \quad \textit{and} \quad \mathsf{FAct}(\phi_1 \wedge \phi_2) = \mathsf{FAct}(\phi_1) \cup \mathsf{FAct}(\phi_2)$

4. *If* ϕ *is a formula and* $v \in \Gamma_{Var}$ *a variable, then* $\exists_v \phi$ *is a formula.*

   $\mathsf{FV}(\exists_v \phi) = \mathsf{FV}(\phi) \backslash \{v\} \quad \textit{and} \quad \mathsf{FAct}(\exists_v \phi) = \mathsf{FAct}(\phi)$

5. *If* ϕ *is a formula and* $A \in \Gamma_{Act}$ *an action, then* $\exists_A \phi$ *is a formula;* $\mathsf{FV}(\exists_A \phi) = \mathsf{FV}(\phi)$. $\mathsf{FAct}(\exists_A \phi) = \mathsf{FAct}(\phi) \backslash \{A\}$

Notation: Let ϕ, ϕ_1 and ϕ_2 denote first-order logic formulas and let p denote a propositional constant. Then the following common shorthands will be used in the sequel ($\equiv$ denotes syntactical equality):

$$\begin{array}{lcllcl}
\phi_1 \vee \phi_2 & \equiv & \neg(\neg\phi_1 \wedge \neg\phi_2) & \forall_v \phi & \equiv & \neg\exists_v \neg\phi \\
\phi_1 \Rightarrow \phi_2 & \equiv & \neg\phi_1 \vee \phi_2 & \forall_A \phi & \equiv & \neg\exists_A \neg\phi \\
\phi_1 \Leftrightarrow \phi_2 & \equiv & (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1) & & & \\
\mathsf{true} & \equiv & (p \vee \neg p) & \mathsf{false} & \equiv & (p \wedge \neg p)
\end{array}$$

If C denotes some finite set, without loss of generality (wlog.) $C = \{c_1, \ldots, c_n\}$, then the following shorthand notations are used:

$$\begin{array}{lcllcl}
\bigwedge\limits_{c \in C} p(c) & \equiv & p(c_1) \wedge \cdots \wedge p(c_n) & \bigwedge\limits_{c \in C} p_c & \equiv & p_{c_1} \wedge \cdots \wedge p_{c_n} \\
\bigwedge\limits_{c \in C} c & \equiv & c_1 \wedge \cdots \wedge c_n & \bigwedge\limits_{1 \le i \le n} c_i & \equiv & c_1 \wedge \cdots \wedge c_n \\
\forall_C\, p & \equiv & \forall_{c_1} \cdots \forall_{c_n}\, p & & &
\end{array}$$

and similarly for $\bigvee$ and $\exists$.

To dissolve ambiguities in a flat representation, any (sub)formula may be enclosed in parenthesis. For example, both $\exists_v(\phi)$ and $(\exists_v \phi)$ might be used. Besides, the following binding order of operators is assumed:

$$\begin{array}{ccccccc}
\Rightarrow & & & & \forall & & \\
\Leftrightarrow & \vee & \wedge & \neg & \exists & = &
\end{array}$$

Thus, $(\forall_x \neg(x = 0)) \wedge (y = 1)$ may be simplified to $\forall_x \neg x = 0 \wedge y = 1$.

Finally, Boolean valued variables will be used frequently. This means, there is a sort Boolean and two constants of this sort. In abuse of notation, these constants will also be called *true* and *false*. Then, $b =$*true* and $b =$*false* are abbreviated by b and $\neg b$. As the same notation is used for propositional constants, we take care that from the context it is clear whether b is a Boolean valued variable or a propositional constant[2].

In the context of the TLT programming notation, it is convenient to distinguish so-called state predicates as a special class of first-order logic formulas. They are used to describe the states of a specification and are joined by transition predicates used to formalize transitions.

Definition 3.1.5 (State and Transition Predicates)
A state predicate ϕ is a formula as defined above, but such that $\mathsf{FAct}(\phi) = \emptyset$ *and such that* $\mathsf{FV}(\phi) \subseteq \Gamma_V \cup \Gamma_S$.

First-order logic formulas as defined above are also called transition predicates.[3]

3.1.2 Semantics

In this section, the first-order logic formulas introduced above are given a formal meaning, called the semantics of the formulas. The definition of the semantics follows the inductive structure of the syntax of the formulas. The basis is laid by fixing the meaning of the symbols occurring in the signature of the first-order logic. Whereas for the sort, function and predicate symbols an *interpretation* is fixed, the variable and action symbols are *evaluated*. This separation of interpretation and evaluation reflects the special role of place holders that variables and actions play.

Definition 3.1.6 (Interpretation $\mathcal{I}$)
Let $\mathcal{U}$ be a set $\mathcal{U} \supseteq \{\surd\}$, called (common) universe.

A first-order logic interpretation $\mathcal{I}$ maps

1. *elementary sort names $S \in \Gamma_{Sort}$ to nonempty subsets D of the universe $\mathcal{U}$, that is, $\mathcal{I}(S) = D \subseteq \mathcal{U}$. As a special case, the unit sort is mapped to $\{\surd\}$, that is, $\mathcal{I}(()) = \{\surd\}$. Furthermore, let $S_1, S_2 \in \Gamma_{Sort}$ and $D_i = \mathcal{I}(S_i)$. Then $\mathcal{I}(S_1 \times S_2) \stackrel{def}{=} D_1 \times D_2$.*

2. *n-ary function symbols $f \in \Gamma_{Func}$ with $\mathrm{arity}(f) = (S_1, \ldots, S_n)$ and $\mathrm{sort}(f) = S$ to functions $\mathcal{I}(f) = (D_1 \times \cdots \times D_n \to D)$ with $D_i = \mathcal{I}(S_i)$ and $D = \mathcal{I}(S)$.*
   As a special case, constant symbols c with $\mathrm{sort}(c) = S$ are mapped to functions $\mathcal{I}(c) = (\{\surd\} \to D)$ that are identified with elements of $D = \mathcal{I}(S)$.

3. *n-ary predicate symbols $p \in \Gamma_{Pred}$ with $\mathrm{arity}(p) = (S_1, \ldots, S_n)$ to n-ary relations $\mathcal{I}(p) \subseteq (D_1 \times \cdots \times D_n)$ where $D_i = \mathcal{I}(S_i)$.*
   As a consequence, propositional constants are mapped to either $\emptyset$ or $\{\surd\}$;

[2]However, Boolean valued variables should not be confused with propositional constants. The meaning of the former depends on a valuation (that may change over time) whereas the meaning of a propositional constant is fixed once and for all for the whole system under consideration.

[3]Whereas in the sequel "formula" will be used synonymously for "temporal logic formula".

For all specification variables $c \in \Gamma_S$, we require that the set of closed terms of sort $\mathrm{sort}(c)$ *is effectively computational and large enough to name all elements of the corresponding* $\mathcal{I}(\mathrm{sort}(c))$, *i.e. for all* $u \in \mathcal{I}(\mathrm{sort}(c))$ *there is a closed term* t *of sort* $\mathrm{sort}(c)$ *such that* $\mathcal{I}(t) = u$[4].

In the sequel, $\mathcal{D}(a)$ *abbreviates* $\mathcal{I}(\mathrm{sort}(a))$ *for any variable or action* a. $\mathcal{D}(a)$ *is called the domain of* a.

Usually, the same notation is used for constant symbols and their interpretation. For example, $\mathcal{I}(true)$=*true* and $\mathcal{I}(false)$=*false* for the Boolean constants.

According to the separation of the variable symbols into disjoint sets of primed, unprimed and specification variables, the valuation of the variables is divided into three valuation functions. Together with the valuation function for action symbols, the valuation function is thus split into four parts.

Definition 3.1.7 (Valuation)
Given an interpretation $\mathcal{I}$, *a valuation* $\beta \stackrel{def}{=} (\gamma, \zeta, \alpha, \zeta')$ *of the variables* Γ_{Var} *and actions* Γ_{Act} *is a function* $\beta : \Gamma_{Var} \cup \Gamma_{Act} \to \mathcal{U} \dot\cup \{\bot\}$ *defined as:*

$$\beta(a) \stackrel{def}{=} \begin{cases} \gamma(a) & , a \in \Gamma_S \\ \zeta(a) & , a \in \Gamma_V \\ \alpha(a) & , a \in \Gamma_{Act} \\ \zeta'(a) & , a \in \Gamma_{V'} \end{cases}$$

where $\gamma, \zeta, \alpha,$ *and* ζ' *are functions*

- $\gamma : \Gamma_S \to \mathcal{U}$
  $\gamma : x \mapsto \gamma(x) \in \mathcal{D}(x)$;
- $\zeta : \Gamma_V \to \mathcal{U}$
  $\zeta : x \mapsto \zeta(x) \in \mathcal{D}(x)$;
- $\alpha : \Gamma_{Act} \to \mathcal{U} \cup \{\bot\}$
  $\alpha : A \mapsto \alpha(A) \in \mathcal{D}(a) \cup \{\bot\}$;
- $\zeta' : \Gamma_{V'} \to \mathcal{U}$
  $\zeta' : x' \mapsto \zeta'(x') \in \mathcal{D}(x')$;

For actions a, *the set* $\mathcal{D}_\bot(a) \stackrel{def}{=} \mathcal{D}(a) \cup \{\bot\}$ *is called extended domain of* a. *Let* $\Gamma_1 \subseteq \Gamma_{Var} \cup \Gamma_{Act}$. *Then* $\beta|_{\Gamma_1}$ *denotes the function* $\beta|_{\Gamma_1} : \Gamma_1 \to \mathcal{U} \dot\cup \{\bot\}$ *defined by* $\beta|_{\Gamma_1}(a) = \beta(a)$ *for all* $a \in \Gamma_1$.

In this definition, the valuation of actions differs from the valuation of variables in letting $\bot$ be a possible value for actions. This reflects the case where an action does not occur, that is, no synchronization takes place (and no value is communicated). To simplify reference, if $\alpha(A) \in \mathcal{D}(A)$, we say that A "occurs" and that it has the value $\alpha(A)$. If $\alpha(A) = \bot$, we say that A "does not occur".

Now subsequently the semantics of terms and formulas based on an interpretation and a valuation are defined.

The semantics of a term is an element in the subset of the universe that corresponds to the sort of the term. More formally, given an interpretation and a valuation, the semantics $[\![\ ]\!]$ is a function that maps terms $\mathtt{t}$ to elements of $\mathrm{sort}(\mathtt{t})$, that is, $[\![\ ]\!] : \mathtt{t} \mapsto [\![\mathtt{t}]\!] \in \mathrm{sort}(\mathtt{t})$.

[4]Definition 3.1.8 below extends $\mathcal{I}$ to arbitrary closed terms.

Unprimed variables, actions and primed variables represent the current state of a specification, the synchronization and communication while passing to the next-state, and the next-state respectively. On the other hand, specification variables will be used for global parameters of systems (like a lift with "n" floors) as well as for writing schemata of conditions, commands and transitions. In this latter usage, the specification variables will always be bounded. Thus, free specification variables will only occur on a global system specification level and therefore have more in common with the interpretation than with the flexible variables. This difference will be made explicit in the notation we will use: the valuation function γ for specification variables is placed on the same level as the interpretation $\mathcal{I}$.

Definition 3.1.8 (Semantics of Terms)
Given an interpretation $\mathcal{I}$ and a valuation $(\gamma, \zeta, \alpha, \zeta')$, the semantics $[\![t]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$ of a term t is defined by:

$$[\![t]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} \stackrel{def}{=} \begin{cases} \gamma(t) & \text{for } t \in \Gamma_S, \\ \zeta(t) & \text{for } t \in \Gamma_V, \\ \zeta'(t) & \text{for } t \in \Gamma_{V'}, \\ \mathcal{I}(f)([\![t_1]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}, \ldots, [\![t_n]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}) & \text{for } t \equiv f(t_1, \ldots, t_n), \\ [\![t_1]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} \times [\![t_2]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} & \text{for } t \equiv (t_1, t_2) \end{cases}$$

The semantics of atomic formulas is a function that maps the formulas to their truth value. In the definition below the semantics are given for the "true" case only, that is, $[\![\phi]\!]$ should be read as "ϕ is true".

Definition 3.1.9 (Semantics of Atomic Formulas)
Given an interpretation $\mathcal{I}$ and a valuation $(\gamma, \zeta, \alpha, \zeta')$, the semantics of atomic formulas is defined by:

$[\![p(t_1, \ldots, t_n)]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$ **iff** $([\![t_1]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}, \ldots, [\![t_n]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}) \in \mathcal{I}(p)$
for a predicate $p(t_1, \ldots, t_n)$

$[\![A(t)]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$ **iff** $\alpha(A) = [\![t]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$,
for an action A and a term t

$[\![t_1 = t_2]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$ **iff** $[\![t_1]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} = [\![t_2]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$
for terms t_1 and t_2

The intended meaning of $[\![\phi]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$ is "the formula ϕ is true under interpretation $\mathcal{I}$ and valuation $(\gamma, \zeta, \alpha, \zeta')$".

Finally, the semantics of (first-order logic) formulas also contains a definition for existential quantification over actions, which has to include the possibility that an action does not happen.

Definition 3.1.10 (Semantics of Formulas)
Given a first-order logic interpretation $\mathcal{I}$ and a valuation $(\gamma, \zeta, \alpha, \zeta')$, the semantics of formulas is inductively defined:

If ϕ is an atomic formula, then its semantics is defined in Definition 3.1.9; otherwise:

$$[\![\neg\phi]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} \quad \textbf{iff} \quad \textit{not } [\![\phi]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$$

$$[\![\phi_1 \wedge \phi_2]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} \quad \textbf{iff} \quad [\![\phi_1]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} \textit{ and } [\![\phi_2]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$$

$$[\![\exists_x \phi]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} \quad \textbf{iff} \quad \begin{cases} [\![\phi]\!]^{(\zeta[x\leftarrow d],\alpha,\zeta')}_{\mathcal{I},\gamma} & \textit{for some } d \in \mathcal{D}(x) \quad , \textit{if } x \in \Gamma_V \\ [\![\phi]\!]^{(\zeta,\alpha,\zeta'[x\leftarrow d])}_{\mathcal{I},\gamma} & \textit{for some } d \in \mathcal{D}(x) \quad , \textit{if } x \in \Gamma_{V'} \\ [\![\phi]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma[x\,\leftarrow\, d]} & \textit{for some } d \in \mathcal{D}(x) \quad , \textit{if } x \in \Gamma_S \end{cases}$$

$$[\![\exists_A \phi]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} \quad \textbf{iff} \quad [\![\phi]\!]^{(\zeta,\alpha[A\leftarrow d],\zeta')}_{\mathcal{I},\gamma} \quad \textit{for some } d \in \mathcal{D}(A) \cup \{\bot\};$$

$$\textit{where } \beta[a \leftarrow d](b) \stackrel{def}{=} \begin{cases} \beta(b) & , b \neq a \\ d & , b = a \end{cases} \quad \textit{for valuations } \beta.$$

As a consequence of free specification variables occurring only on system level, their valuation γ as well as the interpretation $\mathcal{I}$ are usually fixed for a specification under consideration. Thus the subscripts $_{\mathcal{I},\gamma}$ will often be omitted.

With $\mathcal{I}$ and γ being fixed,

- *the valuation (ζ, α, ζ') is called a model of ϕ* **iff** $[\![\phi]\!]^{(\zeta,\alpha,\zeta')}$,
- *ϕ is called satisfiable* **iff** *there is a valuation (ζ, α, ζ') that is a model of ϕ, and*
- *ϕ is called valid* **iff** *all valuations (ζ, α, ζ') are models of ϕ.*

REMARK 1. true and false.

The symbols true and false are (ab)used several times. As syntactic entity, true names a Boolean constant and abbreviates the formula $(p \vee \neg p)$. As a semantic entity, true is an element of the universe.

As a Boolean constant, it might be used together with a Boolean valued variable x in a formula like $x =$ true. Then, $[\![x = \mathsf{true}]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$ **iff** $\zeta(x) = \mathcal{I}(\mathsf{true}) = \mathsf{true}$.

Used as an abbreviation of $(p \vee \neg p)$, true is a valid formula for any interpretation $\mathcal{I}$ and any valuation of the specification variables γ.

REMARK 2. Comparison of actions in TLT versus actions in CCS.

In contrast to pure CCS, actions in TLT are typed. This has advantages if there are only finitely many actions (channels) but infinite data types. In such situations it is often straightforward to abstract from the actual data sent resulting in finite (control) specifications.

On the other hand, in TLT there is no notion of an algebra of actions. Instead, actions are used exclusively to have values assigned to them.

Examples.

Let A be a Boolean valued action and let x be a Boolean valued variable. The following table shows the truth values of some formulas for all possible valuations of A and x.

Valuation of A	Valuation of x	$A(x)$	$\exists_x A(x)$	$\exists_A A(x)$
true	true	is true	is true	is true
	false	is false	is true	is true
false	true	is false	is true	is true
	false	is true	is true	is true
$\perp$	true	is false	is false	is true
	false	is false	is false	is true

From the table it follows that $\exists_A A(x)$ is valid. Other valid formulas for this example are $\forall_x \exists_A A(x)$, $\exists_x \exists_A A(x)$ and $\exists_A \exists_x A(x)$. There are also formulas that are not satisfiable like, for example, $\forall_A \exists_x A(x)$, $\forall_A \forall_x A(x)$, $\forall_x \forall_A A(x)$, $\exists_x \forall_A A(x)$ and $\exists_A \forall_x A(x)$.

In the following table sample formulas are evaluated for various given sorts and valuations of actions and variables. The sort of an action or variable a is written using the notation a : sort(a) for declarations formally introduced in Section 5.3.

Declaration	Valuation	(ζ, α, ζ') is a model for	(ζ, α, ζ') is **not** a model for
A : ()	$\alpha(A) = \surd$	A	$\neg A$
	$\alpha(A) = \perp$	$\neg A$	A
	$\alpha(A)$ arb.	$\exists_A\ A$	$\forall_A\ A$
A : {1,2}	$\alpha(A) = 1$	$A(1)$ $\neg A(2)$ $\exists_c A(c)$	$\neg A(1)$ $A(2)$
	$\alpha(A) = \perp$	$\neg A(1)$ $\neg A(1) \wedge \neg A(2)$	$A(1)$ $\exists_c A(c)$
	$\alpha(A)$ arb.	$\neg A(1) \vee \neg A(2)$	$A(1) \wedge A(2)$
A : S x : S	$\alpha(A)$ arb. $\zeta(x)$ arb.	$\exists_A\ A(x)$ $\exists_A\ \neg A(x)$ $\forall_c \exists_A A(c)$	$\forall_A\ A(x)$ $\forall_A\ \neg A(x)$ $\exists_A \forall_c A(c)$ $\exists_c \forall_A A(c)$ $\forall_A \exists_c A(c)$
A : () B : ()	$\alpha(A)$ arb. $\alpha(B)$ arb.	$\forall_A\ \exists_B\ (A \Rightarrow B)$	

Lemma 3.1.11 $\quad [\![\forall_x \neg A(x)]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$ **iff** $\alpha(A) = \perp$

Proof:

Recall that $\forall_x \neg A(x)$ to be well-formed requires that $\mathcal{D}(A) = \mathcal{D}(x)$. Wlog. assume $x \in \Gamma_S$ (the proof for $x \in \Gamma_V$ or $x \in \Gamma_{V'}$ is analog).

$[\![\forall_x \neg A(x)]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$

iff $[\![\neg A(x)]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma[x \leftarrow d]}$ for all $d \in \mathcal{D}(x)$

iff not $[\![A(x)]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma[x \leftarrow d]}$ for all $d \in \mathcal{D}(x)$

iff not $\alpha(A) = [\![x]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma[x \leftarrow d]}$ for all $d \in \mathcal{D}(x)$

iff not $\alpha(A) = d$ for all $d \in \mathcal{D}(x) = \mathcal{D}(A)$

iff $\alpha(A) = \perp$ ■

Action quantifiers facilitate the formulation of consistency conditions for TLT specifications. However, they do not add additional expressiveness to first-order logic. This is shown by the following lemma that describes how to "implement" actions by introducing a fresh variable for each action symbol[5]. The action quantifiers may then be reduced to "ordinary" ones as is shown in the subsequent theorem. Another way of implementing actions is described in [Mer95]. There, two variables replace each action: one variable of the same sort as the action carries the value of the action whereas a second Boolean valued variable indicates the presence or absence of the action.

Lemma 3.1.12
Assume that action A does not occur bound in ϕ and let $x \in \Gamma_S$ be a variable not occurring in ϕ and such that $\mathrm{sort}(A) = \mathrm{sort}(x)$. *Further, let $\phi_{[A(t)\leftarrow x=t]}$ denote ϕ with all occurrences $A(t)$ being (syntactically) replaced by $x = t$. Then*

1. $[\![\, \phi \,]\!]^{(\zeta,\alpha[A\leftarrow\perp],\zeta')}_{\mathcal{I},\gamma}$ **iff** $[\![\, \phi_{[A(t)\leftarrow \mathsf{false}]} \,]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$
2. $[\![\, \phi \,]\!]^{(\zeta,\alpha[A\leftarrow d],\zeta')}_{\mathcal{I},\gamma}$ **iff** $[\![\, \phi_{[A(t)\leftarrow x=t]} \,]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma[x\,\leftarrow\, d]}$

Proof of 1. : Induction on syntax of formulas

1. Base case : atomic formulas
   Let $\phi \in \Gamma_{Pred}$, $\phi \equiv t_1 = t_2$, or $\phi \in \Gamma_{Act}$ but such that $\phi \not\equiv A(t)$. Then trivially
   $$[\![\, \phi \,]\!]^{(\zeta,\alpha[A\leftarrow\perp],\zeta')}_{\mathcal{I},\gamma} \quad \textbf{iff} \quad [\![\, \phi \,]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} \quad \textbf{iff} \quad [\![\, \phi_{[A(t)\leftarrow \mathsf{false}]} \,]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$$
   holds.
   For $\phi \in \Gamma_{Act}$ and $\phi \equiv A(t)$, we get

   $[\![\, A(t) \,]\!]^{(\zeta,\alpha[A\leftarrow\perp],\zeta')}_{\mathcal{I},\gamma}$

   iff $\alpha[A \leftarrow \perp](A) = [\![\, t \,]\!]^{(\zeta,\alpha[A\leftarrow\perp],\zeta')}_{\mathcal{I},\gamma}$

   iff $\perp = [\![\, t \,]\!]^{(\zeta,\alpha[A\leftarrow\perp],\zeta')}_{\mathcal{I},\gamma}$

   iff (as terms never evaluate to $\perp$)

   $[\![\, \mathsf{false} \,]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$

   iff $[\![\, A(t)_{[A(t)\leftarrow \mathsf{false}]} \,]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$

2. Induction step :
   As an example let $\phi \equiv \exists_B\ \psi$. Then

   $[\![\, \phi \,]\!]^{(\zeta,\alpha[A\leftarrow\perp],\zeta')}_{\mathcal{I},\gamma}$

   iff $[\![\, \psi \,]\!]^{(\zeta,\alpha[A\leftarrow\perp,B\leftarrow d],\zeta')}_{\mathcal{I},\gamma}$ for some $d \in \mathcal{D}(B) \cup \{\perp\}$

   iff (by inductive hypothesis)

   $[\![\, \psi_{[A(t)\leftarrow \mathsf{false}]} \,]\!]^{(\zeta,\alpha[B\leftarrow d],\zeta')}_{\mathcal{I},\gamma}$ for some $d \in \mathcal{D}(B) \cup \{\perp\}$

   iff $[\![\, \phi_{[A(t)\leftarrow \mathsf{false}]} \,]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$

The proof for the second proposition is analog. ■

[5] This has to be done in order to use standard first-order logic theorem provers.

Theorem 3.1.13 (Syntactic Removal of actions)
Let $x \in \Gamma_S$ be a variable not occurring in ϕ and such that $\mathrm{sort}(A) = \mathrm{sort}(x)$. Further, let $\phi_{[A(t) \leftarrow d]}$ denote ϕ with all (with respect to A free) occurrences $A(t)$ being (syntactically) replaced by d for any t. Then

$$
\begin{array}{ll}
 & [\![\exists_A \ \phi]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} \\
\textbf{iff} & [\![\exists_x \ \phi_{[A(t) \leftarrow x=t]} \ \vee \ \phi_{[A(t) \leftarrow \mathsf{false}]}]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}
\end{array}
$$

$$
\begin{array}{ll}
 & [\![\forall_A \ \phi]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} \\
\textbf{iff} & [\![\forall_x \ \phi_{[A(t) \leftarrow x=t]} \ \wedge \ \phi_{[A(t) \leftarrow \mathsf{false}]}]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}
\end{array}
$$

Proof : First equivalence

$$
\begin{array}{lll}
 & [\![\exists_A \ \phi]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} & \\
\textbf{iff} & [\![\phi]\!]^{(\zeta,\alpha[A \leftarrow d],\zeta')}_{\mathcal{I},\gamma} & \text{for some } d \in \mathcal{D}(A) \cup \{\bot\} \\
\textbf{iff} & [\![\phi]\!]^{(\zeta,\alpha[A \leftarrow d],\zeta')}_{\mathcal{I},\gamma} & \text{for some } d \in \mathcal{D}(A) \ \text{ or } \ [\![\phi]\!]^{(\zeta,\alpha[A \leftarrow \bot],\zeta')}_{\mathcal{I},\gamma} \\
\textbf{iff} & \text{(Lemma 3.1.12)} & \\
 & [\![\phi_{[A(t) \leftarrow x=t]}]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma[x \leftarrow d]} & \text{for some } d \in \mathcal{D}(A) \qquad \text{or} \qquad [\![\phi_{[A(t) \leftarrow \mathsf{false}]}]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} \\
\textbf{iff} & [\![\exists_x \ \phi_{[A(t) \leftarrow x=t]}]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma[x \leftarrow d]} & \text{or} \qquad [\![\phi_{[A(t) \leftarrow \mathsf{false}]}]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma} \\
\textbf{iff} & [\![\exists_x \ \phi_{[A(t) \leftarrow x=t]} \ \vee \ \phi_{[A(t) \leftarrow \mathsf{false}]}]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma[x \leftarrow d]} &
\end{array}
$$

The proof of the equivalence for universal quantification is similar. ■

The theorem states, that action quantifiers are nothing more than syntactic sugar and can be replaced by ordinary quantification. However the resulting formulas loose their "intuitive" semantics. In essence, the action $\exists$-quantor gets replaced by the disjunction of an ordinary $\exists$-quantor and a term describing empty channels. Due to the theorem it is straightforward to reprove for example the following rules, well known for ordinary quantification.

Corollary 3.1.14

$$\exists_a \exists_b \phi \ \equiv \ \exists_b \exists_a \phi \qquad \textit{with } a, b \in \Gamma_{Var} \cup \Gamma_{Act}$$

$$\forall_a \forall_b \phi \ \equiv \ \forall_b \forall_a \phi \qquad \textit{with } a, b \in \Gamma_{Var} \cup \Gamma_{Act}$$

3.2 A Temporal Logic

Historically, temporal logics emerged from modal logics originally introduced by philosophers to reason about systems of connected "worlds" that vary in the set of assertions that are true in them. Thus, each world has a distinct "mode" of truth (therefore "modal" logics). Two predominant operators in these logics are $\Box$ and $\Diamond$. $\Box\mathsf{p}$ is said to hold in a given world w, written $\mathsf{w} \models \Box\mathsf{p}$, if the assertion p is true in *all* worlds reachable from w. Similarly, $\Diamond\mathsf{p}$ is said to hold in a given world w, if the assertion p is true in *some* world reachable from w. Temporal logics are obtained from modal logics by replacing the general notion of connected worlds by a totally ordered set of points (sometimes also intervals) in time. $\mathsf{t} \models \Box\mathsf{p}$ then means that assertion p is true in all points of time in the future of the given point t, or simply, that p holds *always*

in the future of t. $t \models \Diamond p$ then means that assertion p is true in some future point, or simply, that p holds *sometimes* or *eventually* in the future of t.

Summarized, in a temporal logic with the operators $\Box$ and $\Diamond$, a *sequence* of situations may be described and reasoned about. Therefore these logics are called linear time temporal logics. Examples for linear time temporal logics are UNITY (see [CM88]) or TLA (see [Lam94a]). Besides, there are also so-called branching time temporal logics. In branching time temporal logics, each point of time may have several successors, resulting in a *tree* of situations that may be described and reasoned about (see for example [BAPM83]). The best known representative of these logics is CTL, the Computation Tree Logic, as presented in [CE81] and [CES83]. Branching time temporal logics allow to express that a property holds in some possible future whereas in linear time temporal logics valid properties always have to hold for all possible futures, that is, for all sequences. On the other hand, there are properties like strong fairness that are expressible in linear time temporal logics but not in CTL. For this reason there was an ongoing discussion in the eighties about which kind of temporal logic to prefer mainly with respect to expressiveness and the computational complexity to check the satisfiability of formulas. As a result, more expressive logics (like CTL*) were introduced that contained both CTL and linear time temporal logics. A second result of these discussions is a wide variety of articles comparing linear time and branching time temporal logics as, for example, [Lam80], [EL85], [EH86], [BCG88], and [Sti92].

In this thesis, I only consider linear time temporal logics because I personally feel that they are more intuitive to use.

Good introductions to temporal logics may be found in [Eme90] or in the comprehensive standard textbook for linear time temporal logics [MP91], lately expanded by a second volume [MP96]. In [BA93] both first-order logic and temporal logic are dealt in an introductory manner.

The temporal logic presented below subsumes the first-order logic defined above. Following the philosophy of TLA (see [Lam91]), a temporal 'next'-operator is excluded, but since transition predicates can be expressed in the first-order logic anyway, this is no real loss of expressive power. Excluding the 'next'-operator allows for dealing with safety properties and most assumptions within first-order logic. Furthermore, in refining a specification, typically extra steps are introduced that falsify properties containing a temporal 'next'-operator. In contrast to TLA, however, I make no use of short hand notations that implicitly add a frame to formulas, but stay with plain definitions, similar to the ones found, for example, in [MP81].

Whereas most temporal logics found in the literature are based on a propositional logic, the TLT logic is based on a first-order logic. Therefore, an immediate question is whether there should be quantification on the level of the temporal logic. In fact, this turns out to be necessary in order to "hide" local variables of specifications which is essential for both composition and refinement. Indeed, besides the quantification introduced for the first-order logic, there will be two more possibilities to express quantification. Firstly, there is a straightforward extension permitting the quantification of specification variables also for temporal formulas. As said before, specification variables are used bounded for schematas and free for global parameters of the specification under consideration. In the latter case, their value does not change over time. For free occurrences of actions and flexible variables, that may change over time, a second kind of "temporal" quantification is needed. Intuitively, the existential quantification on a flexible variable x will express "there is a sequence of values for x".

As was the case for the first-order logic, the formal definition of the temporal logic is given for a minimal set of operators only. Additional operators are defined as abbreviations.

Definition 3.2.1 (Syntax of Temporal Formulas)
The set of temporal formulas is defined inductively as:

1. *Any first-order logic formula ϕ (see Definition 3.1.10) is a temporal formula Φ.*
   $\mathsf{FV}(\Phi) = \mathsf{FV}(\phi) \qquad \mathsf{FAct}(\Phi) = \mathsf{FAct}(\phi)$
2. *If Φ is a temporal formula, then $\Box\Phi$ ('always') is a temporal formula.*
   $\mathsf{FV}(\Box\Phi) = \mathsf{FV}(\Phi) \qquad \mathsf{FAct}(\Box\Phi) = \mathsf{FAct}(\Phi)$
3. *If Φ is a temporal formula, then $\neg\Phi$ is a temporal formula.*
   $\mathsf{FV}(\neg\Phi) = \mathsf{FV}(\Phi) \qquad \mathsf{FAct}(\neg\Phi) = \mathsf{FAct}(\Phi)$
4. *If Φ_1, Φ_2 are temporal formulas, then $\Phi_1 \wedge \Phi$ is a temporal formula*
   $\mathsf{FV}(\Phi_1 \wedge \Phi_2) = \mathsf{FV}(\Phi_1) \cup \mathsf{FV}(\Phi_2) \qquad \mathsf{FAct}(\Phi_1 \wedge \Phi_2) = \mathsf{FAct}(\Phi_1) \cup \mathsf{FAct}(\Phi_2)$
5. *If Φ is a temporal formula, $s \in \Gamma_S$ and $a \in \Gamma_V \cup \Gamma_{Act}$, then $\exists_s \Phi$ and $\mathbf{\exists}_a \Phi$ are temporal formulas.*
   $\mathsf{FV}(\exists_s \Phi) = \mathsf{FV}(\Phi) - \{s\} \qquad \mathsf{FAct}(\exists_s \Phi) = \mathsf{FAct}(\Phi)$
   $\mathsf{FV}(\mathbf{\exists}_a \Phi) \stackrel{def}{=} \begin{cases} \mathsf{FV}(\Phi) - \{a\} & , a \in \Gamma_V \\ \mathsf{FV}(\Phi) & , a \in \Gamma_{Act} \end{cases} \qquad \mathsf{FAct}(\mathbf{\exists}_a \Phi) \stackrel{def}{=} \begin{cases} \mathsf{FAct}(\Phi) & , a \in \Gamma_V \\ \mathsf{FAct}(\Phi) - \{a\} & , a \in \Gamma_{Act} \end{cases}$

Notation: Let Φ denote a temporal formula, ϕ and ψ denote first-order logic formulas, and p and q denote state predicates. Then the following shorthands (commonly known as 'eventually', 'leads-to', 'weak fair', 'strong fair', 'unless', and 'until respectively) will be used:

$$\begin{array}{lcllcl} \Diamond\Phi & \equiv & \neg\Box\neg\Phi & \mathcal{WF}(\phi,\psi) & \equiv & (\Diamond\Box\phi) \Rightarrow (\Box\Diamond\psi) \\ \phi \longmapsto \psi & \equiv & \Box(\phi \Rightarrow \Diamond\psi) & \mathcal{SF}(\phi,\psi) & \equiv & (\Box\Diamond\phi) \Rightarrow (\Box\Diamond\psi) \\ p \,\mathsf{Unless}\, q & \equiv & \Box(p \wedge \neg q \Rightarrow p' \vee q') & p \,\mathsf{Until}\, q & \equiv & (p \,\mathsf{Unless}\, q) \wedge (p \longmapsto q) \end{array}$$

Furthermore, as usual,

$\forall_s \Phi \equiv \neg\exists_s \neg\Phi \qquad$ and $\qquad \mathbf{\forall}_a \Phi \equiv \neg\mathbf{\exists}_a \neg\Phi$,
$\Phi_1 \vee \Phi_2 \equiv \neg(\Phi_1 \wedge \Phi_2)$, and
$\Phi_1 \Rightarrow \Phi_2 \equiv \neg\Phi_1 \vee \Phi_2 \qquad$ and $\qquad \Phi_1 \Leftrightarrow \Phi_2 \equiv (\Phi_1 \Rightarrow \Phi_2) \wedge (\Phi_2 \Rightarrow \Phi_1)$.

Note that some temporal connectives (like $\wedge$, $\neg$ or $\Rightarrow$) are identical to first-order logic ones.

Finally, the following binding order for the temporal operators is fixed:

$$\begin{array}{ccccccc} & & & & \forall & & \\ & & & & \exists & & \\ \Rightarrow & & & & \mathbf{\forall} & & \\ \Leftrightarrow & \vee & \wedge & \neg & \mathbf{\exists} & = & \\ & & & & \Box & & \\ & & & & \Diamond & & \end{array}$$

In the previous section, the semantics of first-order logic formulas was based on an interpretation and on a valuation of the variables and actions. For temporal formulas, the flexible variables and the actions have to be evaluated for every time instance. Thus, there is now an infinite sequence of valuations for the flexible variables and actions. The valuation of the specification variables as well as the interpretation of sorts, predicates and functions does not change compared to the first-order logic.

More formally, in case of first-order logic, the actions and flexible variables occurring in transition predicates are evaluated by a triple (ζ, α, ζ') of valuation functions for the unprimed variables, the actions, and the primed variables respectively. This models one step or transition where ζ gives the values of the variables "before", ζ' gives the values of the variables "after" the step, and α tells whether some synchronization (and communication) takes place in this step. In order to evaluate the actions and flexible variables over time, one has to consider sequences of such steps, like, for example

$$(\,(\zeta_0, \alpha_0, \zeta_0')\,,\ (\zeta_1, \alpha_1, \zeta_1')\,,\ (\zeta_2, \alpha_2, \zeta_2')\,\ldots\,).$$

These steps are however not independent, as the values "assigned" to the primed variables in step i have to be equal to the values of the corresponding unprimed variables in step $i+1$. That is,

$$\zeta_i'(x') \;=\; \zeta_{i+1}(x) \quad \text{for all } x \in \Gamma_V,$$

which means that ζ_i' and ζ_{i+1} may be identified, resulting in sequences

$$(\,\zeta_0, \alpha_0, \zeta_1, \alpha_1, \zeta_2, \alpha_2, \zeta_3\ \ldots\,) \;\in\; (\,(\Gamma_V \to \mathcal{U}) \times (\Gamma_{Act} \to \mathcal{U} \cup \{\bot\})\,)^\omega$$

of alternating valuations of flexible variables and actions. These are called *valuation sequences* and may also be noted as

$$\zeta_0 \xrightarrow{\alpha_0} \zeta_1 \xrightarrow{\alpha_1} \zeta_2 \xrightarrow{\alpha_2} \zeta_3\ \ldots \qquad .$$

Now the question arises what it means to quantify over an action or flexible variable. Consider, for example, the temporal formula $\exists_y \,\Box\, y' = x+1$ where x and y denote flexible variables. This formula states that there is a sequence of values of y such that in every step ("always") the new value of y equals the old value of x incremented by one. For example, let

$$(\zeta_0(x), \zeta_1(x), \zeta_2(x), \ldots) \;=\; (1, 4, 9, 16, 25, \ldots)\ .$$

If for $(\zeta_0(y), \zeta_1(y), \zeta_2(y), \ldots)$ one chooses

$$(7, 2, 5, 10, 17, \ldots) \quad \text{or} \quad (14, 2, 5, 10, 17, \ldots)$$

then clearly $(\zeta_0, \zeta_1, \zeta_2, \ldots)$ fulfills $\Box\, y' = x + 1$. Actually, the formula $\exists_y \,\Box\, y' = x + 1$ is valid, because by defining

$$\zeta_i(y) \stackrel{\text{def}}{=} \begin{cases} \text{arbitrary} & \text{, for } i = 0 \\ \zeta_{i-1}(x) + 1 & \text{, for } i > 0 \end{cases}$$

it is always possible to fulfill $\Box\, y' = x + 1$.

In order to define temporal quantification formally, it is convenient to relate valuation sequences that agree in the valuation of all but one action or variable:

Definition 3.2.2 (Similar Valuation Sequences)
Two valuation sequences $(\zeta_0^0, \alpha_0^0, \zeta_1^0, \alpha_1^0, \ldots)$ *and* $(\zeta_0^1, \alpha_0^1, \zeta_1^1, \alpha_1^1, \ldots)$ *are called* a-similar *if they only differ in the valuation of* a*, that is, if* $\zeta_i^0(x) = \zeta_i^1(x)$ *and* $\alpha_i^0(A) = \alpha_i^1(A)$ *for all* $x, A \neq a$.

After these preparations, it is now possible to define the semantics of temporal logic formulas:

Definition 3.2.3 (Semantics of Temporal Formulas)
Assume an interpretation $\mathcal{I}$, a valuation γ of the specification variables, and an infinite valuation sequence $\sigma = (\zeta_0, \alpha_0, \zeta_1, \alpha_1, \dots)$ such that the ζ_i are valuations of the flexible variables $x \in \Gamma_V$ and the α_i are valuations of the actions $A \in \Gamma_{act}$.

The fact, that a temporal formula Φ holds at position i of σ is inductively defined by:

$$
\begin{array}{lll}
(\sigma,i) \models_{\mathcal{I},\gamma} \phi & \textbf{iff} & [\![\,\phi\,]\!]_{\mathcal{I},\gamma}^{(\zeta_i,\alpha_0,\zeta_i')} \quad \textit{for a first-order logic formula } \phi \\
 & & \textit{with } \zeta_i'(x') \stackrel{def}{=} \zeta_{i+1}(x) \\
(\sigma,i) \models_{\mathcal{I},\gamma} \Box\Phi & \textbf{iff} & (\sigma,j) \models_{\mathcal{I},\gamma} \Phi \quad \textit{for all } j \geq i \\
(\sigma,i) \models_{\mathcal{I},\gamma} \neg\Phi & \textbf{iff} & \textit{not} \quad (\sigma,i) \models_{\mathcal{I},\gamma} \Phi \\
(\sigma,i) \models_{\mathcal{I},\gamma} \Phi_1 \wedge \Phi_2 & \textbf{iff} & (\sigma,i) \models_{\mathcal{I},\gamma} \Phi_1 \textit{ and } (\sigma,i) \models_{\mathcal{I},\gamma} \Phi_2 \\
(\sigma,i) \models_{\mathcal{I},\gamma} \exists_s \Phi & \textbf{iff} & (\sigma,i) \models_{\mathcal{I},\gamma[s\leftarrow d]} \Phi \quad \textit{for some } d \in \mathcal{I}(\mathrm{sort}(s)) \\
(\sigma,i) \models_{\mathcal{I},\gamma} \mathbf{\exists}_a \Phi & \textbf{iff} & (\tau,i) \models_{\mathcal{I},\gamma} \Phi \quad \textit{for some } \tau \textit{ being } a-\textit{similar to } \sigma
\end{array}
$$

Furthermore, $\sigma \models_{\mathcal{I},\gamma} \Phi$ abbreviates $(\sigma,0) \models_{\mathcal{I},\gamma} \Phi$ and is read "Φ holds on σ (with respect to $(\mathcal{I},\gamma)$)". σ is then called model of Φ.

Φ is called satisfiable (with respect to $(\mathcal{I},\gamma)$) **iff** *there is a model σ of Φ.*

Φ is called valid (with respect to $(\mathcal{I},\gamma)$), noted $\models_{\mathcal{I},\gamma} \Phi$ **iff** *all sequences σ are models of Φ.*

Usually one fixed pair $(\mathcal{I},\gamma)$ is used; if this is clear from the context, $\models$ replaces $\models_{\mathcal{I},\gamma}$.

Corollary 3.2.4
The additional notations introduced after Definition 3.2.1 have the following semantics:

$$
\begin{array}{lll}
(\sigma,i) \models_{\mathcal{I},\gamma} \Diamond\Phi & \textbf{iff} & (\sigma,j) \models_{\mathcal{I},\gamma} \Phi \quad \textit{for some } j \geq i; \\
(\sigma,i) \models_{\mathcal{I},\gamma} \phi \longmapsto \psi & \textbf{iff} & \textit{if} \quad [\![\,\phi\,]\!]_{\mathcal{I},\gamma}^{(\zeta_j,\alpha_j,\zeta_j')} \quad \textit{for some } j \geq i \\
 & & \textit{then} \quad [\![\,\psi\,]\!]_{\mathcal{I},\gamma}^{(\zeta_k,\alpha_k,\zeta_k')} \quad \textit{for some } k \geq j \\
(\sigma,i) \models_{\mathcal{I},\gamma} \mathcal{WF}(\phi,\psi) & \textbf{iff} & \textit{for all } j \geq i \textit{ holds :} \\
 & & \quad [\![\,\neg\phi\,]\!]_{\mathcal{I},\gamma}^{(\zeta_k,\alpha_k,\zeta_k')} \quad \textit{for some } k \geq j \\
 & & \quad \textit{or } [\![\,\psi\,]\!]_{\mathcal{I},\gamma}^{(\zeta_k,\alpha_k,\zeta_k')} \quad \textit{for some } k \geq j; \\
(\sigma,i) \models_{\mathcal{I},\gamma} \mathcal{SF}(\phi,\psi) & \textbf{iff} & \textit{for all } j \geq i \textit{ holds :} \\
 & & \quad \textit{if } [\![\,\phi\,]\!]_{\mathcal{I},\gamma}^{(\zeta_k,\alpha_k,\zeta_k')} \quad \textit{for infinitely many } k \geq j \\
 & & \quad \textit{then } [\![\,\psi\,]\!]_{\mathcal{I},\gamma}^{(\zeta_k,\alpha_k,\zeta_k')} \quad \textit{for infinitely many } k \geq j;
\end{array}
$$

Examples.
In [MP91], a lot of theorems for propositional linear temporal logic are listed. For example, let Φ and Ψ denote arbitrary temporal formulas and let ϕ and ψ be transition predicates (that is, first-order logic formulas). Then the following formulas are valid, i.e., they hold for arbitrary valuation sequences σ

1. $\models_{\mathcal{I},\gamma} \Box\,(\Phi \wedge \Psi) \;\Leftrightarrow\; \Box\,\Phi \;\wedge\; \Box\,\Psi$

2. $\models_{\mathcal{I},\gamma} \quad \Box\,\Phi \;\vee\; \Box\,\Psi \quad\Rightarrow\quad \Box\,(\Phi \vee \Psi)$

3. $\models_{\mathcal{I},\gamma} \quad \mathcal{SF}(\phi,\psi) \quad\Rightarrow\quad \mathcal{WF}(\phi,\psi)$

For the latter two formulas, the opposite is not true in general. For example,

$$\Box\,(even(x) \vee odd(x)) \quad\Rightarrow\quad \Box\,even(x) \;\vee\; \Box\,odd(x)$$

does not hold for $\zeta_i(x) \stackrel{\text{def}}{=} i$, whereas

$$\mathcal{WF}(x=0, x=2) \quad\Rightarrow\quad \mathcal{SF}(x=0, x=2)$$

is violated, for example, by the valuation sequence defined by $\zeta_{2*i}(x) \stackrel{\text{def}}{=} 0$ and $\zeta_{2*i+1}(x) \stackrel{\text{def}}{=} 1$.

To state some propositions dealing with (flexible) quantification, some arbitrary $a \in \Gamma_V \cup \Gamma_{Act}$ is fixed. Then, for example, the following propositions hold

1. $\models_{\mathcal{I},\gamma} \quad \Box\forall_a\,\Phi \quad\Leftrightarrow\quad \forall_a\Box\Phi$

2. $\models_{\mathcal{I},\gamma} \quad \Box\forall_a\,\phi \quad\Leftrightarrow\quad \Box\forall_a\phi$

3. $\models_{\mathcal{I},\gamma} \quad \exists_a\Box\Phi \quad\Rightarrow\quad \Box\exists_a\Phi$

To prove the last proposition, one has to show that for some arbitrary valuation sequence σ

$$\sigma \;\models_{\mathcal{I},\gamma}\; \exists_a\Box\Phi \qquad \textbf{implies} \qquad \sigma \;\models_{\mathcal{I},\gamma}\; \Box\exists_a\Phi \quad .$$

Thus suppose $\sigma \;\models_{\mathcal{I},\gamma}\; \exists_a\Box\Phi$. Then,

$$\tau \;\models_{\mathcal{I},\gamma}\; \Box\Phi \qquad \text{for some } a\text{-variant } \tau \text{ of } \sigma.$$

Let τ_0 be such an a-variant of σ, that is,

$$\tau_0 \;\models_{\mathcal{I},\gamma}\; \Box\Phi \quad .$$

Then, by the semantics of $\Box$,

$$(\tau_0, j) \;\models_{\mathcal{I},\gamma}\; \Phi \qquad \text{for all } j \geq 0.$$

Thus,

$$(\;(\tau, j) \;\models_{\mathcal{I},\gamma}\; \Phi \qquad \text{for some } a\text{-variant } \tau \text{ of } \sigma\;) \qquad \text{for all } j \geq 0\;.$$

That is,

$$(\sigma, j) \;\models_{\mathcal{I},\gamma}\; \exists_a\;\Phi \qquad \text{for all } j \geq 0$$

which finally implies

$$\sigma \;\models_{\mathcal{I},\gamma}\; \Box\exists_a\;\Phi\;.$$

The opposite is not true in general. For example,

$$\Box\exists_y\;(\,y = x \;\wedge\; \Box y' = y)$$

is valid, whereas the only models of

$$\exists_y\Box\;(\,y = x \;\wedge\; \Box y' = y)$$

are the valuation sequences $\sigma = (\zeta_i, \alpha_i)_i$ with $\zeta_i(x)$ being constant for all i.

Until now, the semantics of formulas was given with respect to valuation sequences for "all" variables and actions. Obviously, however, it is sufficient, to base the semantics of a formula only on the variables and actions that occur free within the formula. In the programming notation introduced in Chapter 5, the meaning of formulas will be defined in the context of a given set of variable and action declarations. The projection of the models of a formula to these declarations are called the trace of the formula. Formally,

Definition 3.2.5 (Traces)
Let $\mathsf{FFV}(\Phi) \stackrel{def}{=} \mathsf{FV}(\Phi) \cap \Gamma_V$ *denote the free flexible variables of a formula* Φ. *Furthermore, let* $\mathsf{Vars}, \mathsf{Acts}$ *be two sets such that* $\mathsf{FFV}(\Phi) \subseteq \mathsf{Vars} \subseteq \Gamma_V$ *and* $\mathsf{FAct}(\Phi) \subseteq \mathsf{Acts} \subseteq \Gamma_{Act}$. *The set of traces wrt (with respect to)* $\mathsf{Vars}, \mathsf{Acts}$ *is then defined as*

$$\mathbf{Traces}_{\mathsf{Vars},\mathsf{Acts}}(\Phi) \stackrel{def}{=} \{\sigma|_{\mathsf{Vars},\mathsf{Acts}} \mid \sigma \models \Phi \}$$

where $\sigma|_{\mathsf{Vars},\mathsf{Acts}} = (\tilde{\zeta_0}, \tilde{\alpha_0}, \tilde{\zeta_1}, \tilde{\alpha_1}, \dots)$ *with* $\begin{cases} \tilde{\zeta_i} \ : \ \mathsf{Vars} \to \mathcal{U} \\ \tilde{\zeta_i}(x) = \zeta_i(x) \end{cases}$ *and* $\begin{cases} \tilde{\alpha_i} \ : \ \mathsf{Acts} \to \mathcal{U} \cup \{\bot\} \\ \tilde{\alpha_i}(x) = \alpha_i(x) \end{cases}$.

If Vars *and* Acts *are clear from the context,* $\mathbf{Traces}(\Phi)$ *abbreviates* $\mathbf{Traces}_{\mathsf{Vars},\mathsf{Acts}}(\Phi)$.

The following lemma summarizes the most important properties of traces.

Lemma 3.2.6

1. *Let* $\tilde{\sigma} = (\tilde{\zeta_0}, \tilde{\alpha_0}, \tilde{\zeta_1}, \tilde{\alpha_1}, \dots)$ *with* $\begin{cases} \tilde{\zeta_i} \ : \ \mathsf{Vars} \to \mathcal{U} \\ \tilde{\alpha_i} \ : \ \mathsf{Acts} \to \mathcal{U} \cup \{\bot\} \end{cases}$
   Then $\tilde{\sigma} \in \mathbf{Traces}(\Phi)$
   iff $\sigma \models \Phi$ *for all* $\sigma = (\zeta_0, \alpha_0, \dots)$ *with* $\begin{cases} \zeta_i(x) = \tilde{\zeta_i}(x) \text{ for all } x \in \mathsf{Vars} \\ \alpha_i(A) = \tilde{\alpha_i}(A) \text{ for all } A \in \mathsf{Acts} \end{cases}$
2. $\mathbf{Traces}(\Phi_1 \wedge \Phi_2) = \mathbf{Traces}(\Phi_1) \cap \mathbf{Traces}(\Phi_2)$
3. $\models \Phi_1 \Rightarrow \Phi_2$ **iff** $\mathbf{Traces}(\Phi_1) \subseteq \mathbf{Traces}(\Phi_2)$

Proof:

1. By definition, $\mathsf{FFV}(\Phi) \subseteq \mathsf{Vars} \subseteq \Gamma_V$ and $\mathsf{FAct}(\Phi) \subseteq \mathsf{Acts} \subseteq \Gamma_{Act}$. Therefore, the semantics of Φ do not depend on the valuations of variables from $\Gamma_V - \mathsf{Vars}$ and actions from $\Gamma_{Act} - \mathsf{Actions}$.

2. By definition.

3. To see the direction from left to right, suppose $\tilde{\sigma} \in \mathbf{Traces}(\Phi_1)$, i.e. $\sigma \models \Phi_1$ for any σ that agrees on Vars and Acts with $\tilde{\sigma}$. Since $\Phi_1 \Rightarrow \Phi_2$ is valid, especially $\sigma \models \Phi_1 \Rightarrow \Phi_2$, that is, $\sigma \models \Phi_1$ implies $\sigma \models \Phi_2$. Thus $\tilde{\sigma} \in \mathbf{Traces}(\Phi_2)$ by definition.

   For the opposite direction, assume there is some σ such that $\sigma \models \Phi_1$ but not $\sigma \models \Phi_2$. That means $\tilde{\sigma} \in \mathbf{Traces}(\Phi_1)$ but $\tilde{\sigma} \notin \mathbf{Traces}(\Phi_2)$ which contradicts the premise $\mathbf{Traces}(\Phi_1) \subseteq \mathbf{Traces}(\Phi_2)$.

■

In applications, widely known theorems of linear temporal logic (like, for example, the transitivity of leads-to) will be used as needed, but without presenting a full calculus.

Chapter 4

Automata

Besides logic the most commonly used formal models for distributed systems are various kinds of somehow coupled automata. In this chapter, Boolean transition systems as special kind of automata will be presented. In contrast to most other automata, Boolean transition systems are characterized by the edges of the automata being labelled by elements of a Boolean algebra. Therefore, this chapter starts with a brief introduction to Boolean algebras.

4.1 Boolean Algebras

This section summarizes some facts about Boolean algebras that are needed in the following. Most definitions and lemmas can already be found in [Hal74], the standard text book about Boolean algebras, on pages 1-71. The notions of tensor products and independence of Subalgebras are explained in more detail in [Kur94] on pages 31-44. However, in contrast to [Kur94], tensor products are only defined for complete, atomic Boolean algebras. This way, I can give a direct definition instead of the somewhat cumbersome definition in [Kur94] that is based on interior products, monomorphisms and independent subalgebras. I include proofs to make the thesis self-contained. As I do not present the full theory, some proofs are more elementary (and therefore sometimes slightly less elegant) than the corresponding ones in [Hal74] or [Kur94]. A reader familiar with the traditional theory of Boolean algebras may safely skip the first definitions (besides Definition 4.1.12) and start reading in Section 4.1.5.

Definition 4.1.1 (Boolean Algebras)
Let B be a set that contains two distinguished elements 0 and 1, and that is closed under the two binary operations $*$ and $+$ and under the unary operation $^-$.[1] *Then $\mathcal{B} =< B, *, +, ^-, 1, 0 >$ is called a Boolean algebra* **iff** *for arbitrary elements $a, b, c \in B$ the following axioms are fulfilled:*

$a * (b * c) = (a * b) * c$	*associativity*	$a + (b + c) = (a + b) + c$
$a * b = b * a$	*commutativity*	$a + b = b + a$
$a * 1 = a$	*identity rules*	$a + 0 = a$
$a * \bar{a} = 0$	*inverse elements*	$a + \bar{a} = 1$
$a * (b + c) = (a * b) + (a * c)$	*distributivity*	$a + (b * c) = (a + b) * (a + c)$

Sometimes B will be identified with $\mathcal{B}$.

[1] That is $* : B \times B \to B, \ldots$ are total functions.

Corollary 4.1.2

In every Boolean algebra the following equations are true.

$\bar{0} = 1$		$\bar{1} = 0$
$a * 0 = 0$		$a + 1 = 1$
$a * a = a$	*idempotency*	$a + a = a$
$a * (a + b) = a$	*absorption*	$a + (a * b) = a$
$\overline{a * b} = \bar{a} + \bar{b}$	*De Morgan*	$\overline{a + b} = \bar{a} * \bar{b}$

By defining $a \leq b \stackrel{\text{def}}{=} a * b = a$, a partial order on B gets defined, that is, $\leq$ is
(1) reflexive: $a \leq a$,
(2) anti-symmetric: $a \leq b$ and $b \leq a$ imply $a = b$, and
(3) transitive: $a \leq b$ and $b \leq c$ imply $a \leq c$.

Further, $a < b \stackrel{\text{def}}{=} a \leq b \wedge a \neq b$.

Lemma 4.1.3

1. $a * b \leq a$ *and* $a \leq a + b$
2. $a_1 \leq b_1$ *and* $a_2 \leq b_2$ *imply both* $a_1 * a_2 \leq b_1 * b_2$ *and* $a_1 + a_2 \leq b_1 + b_2$,
3. $a \leq b$ **iff** $a * \bar{b} = 0$

Proof:

1. Obvious.
2. Obvious.
3. Suppose $a * b = a$. Then $a * b * \bar{b} = a * \bar{b}$, i.e. $0 = a * \bar{b}$.

   Now suppose $a * \bar{b} = 0$. Then $\overline{a * \bar{b}} = \overline{0}$, i.e. $\bar{a} + b = 1$ and thus $a * (\bar{a} + b) = a * 1$ or equivalently $a * b = a$.

■

Examples.

- $< \mathcal{P}(A), \cap, \cup, \bar{}, A, \emptyset >$
  is a Boolean algebra for any countable set A. The complement for some $X \in \mathcal{P}(A)$ is defined as the set difference with respect to A, that is, $\bar{X} \stackrel{\text{def}}{=} A - X$. The order $\leq$ is ordinary set inclusion $\subseteq$. The validity of the axioms is guaranteed by set theory.
- $< \{A \in \mathcal{P}(\mathbb{Z}) \mid |A| \text{ is finite, or } |\mathbb{Z} - A| \text{ is finite } \}, \cap, \cup, \bar{}, \mathbb{Z}, \emptyset >$
  Again $\leq$ is defined as set inclusion $\subseteq$. A set $A \in \mathcal{P}(\mathbb{Z})$ such that $|\mathbb{Z} - A|$ is finite, is called co-finite. An example of a co-finite set is the set of all integers with absolute values greater than 1: $\{ z \in \mathbb{Z} \mid |z| > 1 \}$. The complement of a finite set is co-finite and vice versa. The intersection or union of two finite (co-finite) sets is again finite(co-finite). The intersection or union of a finite and a co-finite set is again co-finite. Thus all operations are well-defined.

- $< \mathrm{For}_{V/\Leftrightarrow}, \wedge, \vee, \neg, true, false >$
  Here For_V denotes the formulas of propositional logic, defined inductively by
  $p ::= true \mid false \mid v \mid p \wedge p \mid p \vee p \mid \neg p$
  where $v \in V$ denotes a finite set of propositional variables.
  Given For_V, $\mathrm{For}_{V/\Leftrightarrow}$ denotes the set of equivalence classes over For_V based on $\Leftrightarrow$. There are $2^{2^{|V|}}$ equivalence classes that correspond to the $2^{2^{|V|}}$ possible truth tables based on the valuations of the variables in V. The order $\leq$ is implication $\Rightarrow$. The validity of the axioms is given by the usual semantics of $\wedge, \vee, \neg, true$, and $false$. □

Boolean algebras may be visualized as Hasse diagrams that are obtained by arranging the elements of the Boolean algebra in such a way that $a < b$ **iff** a and b are connected and b is above a.

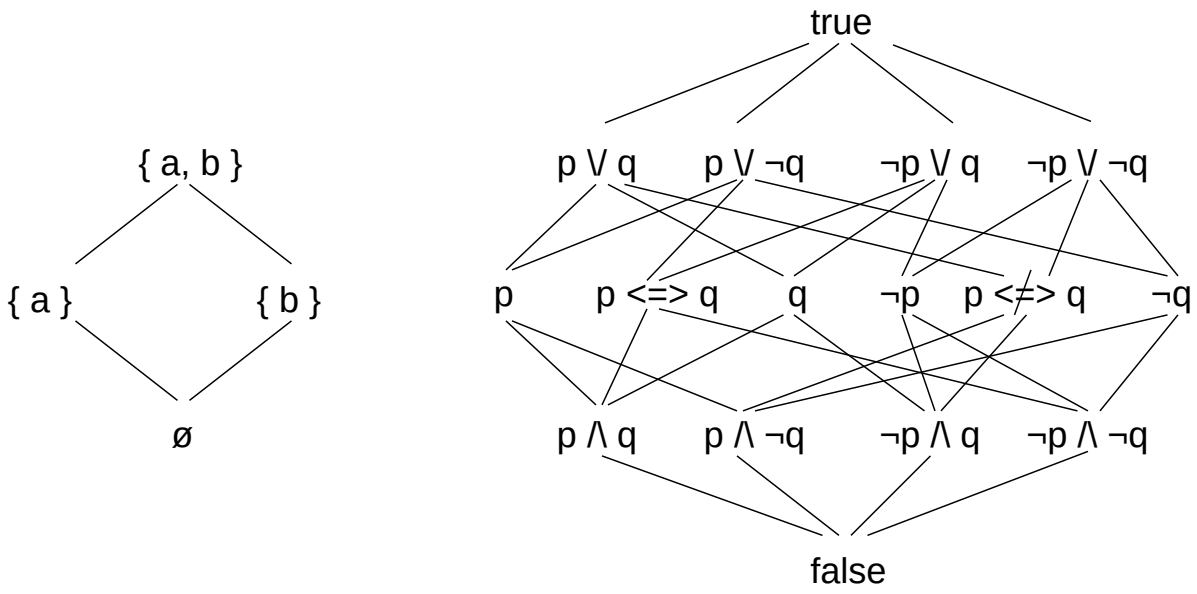


Figure 4.1: The picture shows the Hasse diagrams corresponding to the Boolean algebras $< \mathcal{P}(\{a,b\}), \cap, \cup, ^-, \{a,b\}, \emptyset >$ and $< \mathrm{For}_{\{p,q\}}/\Leftrightarrow, \wedge, \vee, \neg, true, false >$.

4.1.1 Atoms

If one takes a closer look on the examples in Figure 4.1, it becomes apparent that the Boolean algebras presented there are "generated" by the elements in the second row from bottom. The elements drawn above this row always are equivalent to the union (left example) and the disjunction (right example) of the second row elements they dominate. For example, p in the third row is connected to $p \wedge q$ and to $p \wedge \neg q$ and in fact p is equivalent to $p \wedge q \ \vee \ p \wedge \neg q$. This special role of the elements of the second row gives rise to the following definition:

Definition 4.1.4 (Atoms and Atomic Algebras)
An element $s \in B$ is called atom of a Boolean algebra $\mathcal{B}$, **iff** *$s \neq 0$ and for all $b \in B$, $b \leq s$ implies $b = 0$ or $b = s$. $S(\mathcal{B})$ denotes the set of all atoms of $\mathcal{B}$.*
The Boolean algebra $\mathcal{B}$ is called atomic, **iff** *every non-zero element dominates at least one atom, that is,* **iff** *for all $b \in B$ there exists a $s \in S(\mathcal{B})$ such that $s \leq b$.*

Lemma 4.1.5
Let $s \in S(\mathcal{B})$ and $b, b_1, b_2 \in \mathcal{B}$. Then

1. $s \leq b$ *or* $s * b = 0$
2. $s \leq b_1$ *and* $s \leq b_2$ **iff** $s \leq b_1 * b_2$
3. $s \leq b_1$ *or* $s \leq b_2$ **iff** $s \leq b_1 + b_2$

Proof:

1. Firstly, $s * b \leq s$. Since s is an atom, either $s * b = 0$ or $s * b = s$, i.e. either $s * b = 0$ or $s \leq b$.
2. The direction from left to right follows from Lemma 4.1.3. For the opposite direction one has to show $s * b_1 * b_2 = s$ given $s * b_1 = s$ and $s * b_2 = s$. That is obvious.
3. The direction from left to right follows from Lemma 4.1.3. For the opposite direction suppose $s \leq b_1$ does not hold. Then $s * b_1 = 0$ by 1. Thus $s * b_2 = 0 + s * b_2 = s * b_1 + s * b_2 = s * (b_1 + b_2) = s$. That is, $s \leq b_2$.

Lemma 4.1.6
Assume S_1, S_2 to be finite sets of atoms with $S_1, S_2 \subseteq S(\mathcal{B})$, $b_1 = \sum_{s \in S_1} s$ and $b_2 = \sum_{s \in S_2} s$.

If both sums exist, then $b_1 * b_2 = \sum_{s \in S_1 \cap S_2} s$.

Proof:(sketched)

$$b_1 * b_2 = \sum_{s_1 \in S_1} s_1 * \sum_{s_2 \in S_2} s_2 = \sum_{\substack{s_1 \in S_1 \\ s_2 \in S_2}} s_1 * s_2 = \sum_{s \in S_1 \cap S_2} s.$$

The last equation holds, because for atoms $s_1 * s_2 = \begin{cases} s_1 & , \textit{for } s_1 = s_2 \\ 0 & , \textit{otherwise} \end{cases}$. ■

Examples.

- $\mathcal{B} = < \mathcal{P}(A), \cap, \cup, ^-, A, \emptyset >$
  Here $S(\mathcal{B}) = \{ \{a\} \mid a \in A \}$, that is, the atoms are the singletons. Clearly, $\mathcal{B}$ is atomic.
- $< \{A \in \mathcal{P}(\mathbb{Z}) \mid |A|$ is finite, or $|\mathbb{Z} - A|$ is finite $\}, \cap, \cup, ^-, \mathbb{Z}, \emptyset >$
  Again, the atoms of $\mathcal{B}$ are the singletons.
- $\mathcal{B} = < \mathrm{For}_{V/\Leftrightarrow}, \wedge, \vee, \neg, \mathit{true}, \mathit{false} >$ and V is finite
  Here $S(\mathcal{B}) = \{ \bigwedge_I b_i \wedge \bigwedge_J \neg b_j \mid V = \{b_i | i \in I\} \dot\cup \{b_j | j \in J\} \}$.
- $\mathcal{B} = < \mathrm{For}_{V/\Leftrightarrow}, \wedge, \vee, \neg, \mathit{true}, \mathit{false} >$ and V is infinite
  For an infinite set of propositional variables, $\mathrm{For}_{V/\Leftrightarrow}$ is not atomic. Indeed, there are no atoms at all: Let p be an arbitrary formula that is not equivalent to false and let v be a variable not occurring in p. Then $p \wedge v$ does not equal false but $p \wedge v \Rightarrow p$. Thus, p is not an atom. □

4.1.2 Complete Algebras

Definition 4.1.7 (Upper Bounds,Suprema)
An element $u \in B$ is called an upper bound of $A \subseteq B$ **iff** $a \leq u$ *for all $a \in A$. The set of all upper bounds of a subset A of B is denoted by $U(A)$.*
An element $u \in U(A)$ is called supremum of A in $\mathcal{B}$, denoted by $\bigvee A$, **iff** $\bigvee A \leq u$ *for all $u \in U(A)$.*

Dually, lower bounds and infima get defined.

If the supremum of a finite set A exists, then it is unique and equal to the sum of the elements of A:

Lemma 4.1.8
If $\bigvee A$ exists for some finite A, then $\bigvee A = \sum_{a \in A} a$.

Proof:
As $a \leq \sum_{a \in A} a$ for all $a \in A$, $\sum_{a \in A} a$ is an upper bound of A. It remains to show that it is the least upper bound. Thus, let u be an arbitrary upper bound of A, i.e. $a \leq u$ for all $a \in A$. Thus $\sum_{a \in A} a \leq \sum_{a \in A} u = u$. ■

Definition 4.1.9 (Complete Algebras)
A Boolean algebra $\mathcal{B}$ is called complete **iff** $\bigvee A$ *and* $\bigwedge A$ *exist for arbitrary subsets $A \subseteq B$.*

Atomic Boolean algebras have the important property, that all elements are equal to the supremum of the atoms they dominate:

Lemma 4.1.10
Let $\mathcal{B}$ be atomic. Then for arbitrary $b \in \mathcal{B}$

$$b = \bigvee \{s \in S(\mathcal{B}) \mid s \leq b\} \quad .$$

Proof:([Hal74],p.70, lemma 1)

Remark: This implies that the supremum exists (without referring to completeness)!

b is by definition an upper bound of $S_b \stackrel{\text{def}}{=} \{s \in S(\mathcal{B}) \mid s \leq b\}$. Thus it remains to show, that $b \leq u$ for arbitrary upper bounds u of S_b. Suppose not, that is, $b * \bar{u} \neq 0$. Since $\mathcal{B}$ is atomic, there is an atom s_0 such that $0 < s_0 \leq b * \bar{u} \leq b$. In particular, $s_0 \in S_b$, and thus $s_0 * u = s_0$. On the other hand, it also follows $s_0 * u \leq b * \bar{u} * u = 0$ and thus $s_0 * u = 0$ which contradicts $s_0 * u = s_0$. ■

Examples.

- $\mathcal{B} =< \mathcal{P}(A), \cap, \cup, ^-, A, \emptyset >$
  Let X be an element of $\mathcal{P}(A)$, that is, a subset of A. Then all subsets of $\mathcal{P}(A)$ containing X are upper bounds of X. The supremum (and infimum) of X is X.

- $\mathcal{B} =< \mathrm{For}_{V/\Leftrightarrow}, \wedge, \vee, \neg, true, false >$
  Let p be a propositional formula (as representative of a class of equivalent formulas). Then all formulas that imply p are upper bounds. The supremum (and infimum) of p is p.

- $< \{A \in \mathcal{P}(\mathbb{Z}) \mid |A| \text{ is finite, or } |\mathbb{Z} - A| \text{ is finite } \}, \cap, \cup, ^-, \mathbb{Z}, \emptyset >$
  This Boolean algebra is *not* complete. For example, the set of all even integers, $\bigcup_{\substack{z \in \mathbb{Z} \\ even(z)}} \{z\}$, is neither finite nor co-finite. □

4.1.3 Subalgebras

Definition 4.1.11 (Subalgebras)
A Boolean algebra $\mathcal{A} =< A, *, +, ^-, 1, 0 >$ *is called subalgebra of a Boolean algebra* $\mathcal{B} =< B, *, +, ^-, 1, 0 >$ **iff** $A \subseteq B$ *and* A *is closed with respect to the operations* $*$, $+$, *and* $^-$ *of* $\mathcal{B}$ *and shares the* 0 *and* 1 *elements.*

Actually, that $\mathcal{A}$ shares the 0 and 1 elements with $\mathcal{B}$ can already be deduced from the closedness condition: Suppose $a \in A$. Since $\mathcal{A}$ is closed under $^-$, $\bar{a} \in A$, and thus $a * \bar{a} \in A$ and $a + \bar{a} \in A$. But $a * \bar{a} = 0$ and $a + \bar{a} = 1$, i.e. $1, 0 \in A$.

An important special case are subalgebras that do not "interfere" with each other. This is expressed by the property that the product of elements of such *independent* algebras never equals 0 (which can be interpreted as contradiction), given none of the elements already equals 0. This corresponds to TLT specifications with disjoint sets of variables and actions.

Definition 4.1.12
Let $\mathcal{A}_i$ *be a finite set of subalgebras of a Boolean algebra* $\mathcal{B}$. *Then, the* $\mathcal{A}_i$ *are independent* **iff** *for arbitrary* $a_i \in A_i \quad a_1 * \ldots * a_k = 0$ *implies* $a_i = 0$ *for some* i.

4.1.4 Homomorphisms

Morphisms map Boolean algebras to Boolean algebras in a way that preserves important properties.

Definition 4.1.13 (Morphisms)
A homomorphism between two Boolean algebras $\mathcal{A}$ *and* $\mathcal{B}$ *is a function* $\Phi : \mathcal{A} \to \mathcal{B}$ *such that*

- $\Phi(a_1 * a_2) = \Phi(a_1) * \Phi(a_2)$,
- $\Phi(a_1 + a_2) = \Phi(a_1) + \Phi(a_2)$, *and*
- $\Phi(\bar{a}) = \overline{\Phi(a)}$.

If Φ is injective (that is, $\Phi(a_1) = \Phi(a_2)$ implies $a_1 = a_2$), then Φ is called monomorphism, which will be indicated by the notation $\Phi : \mathcal{A} \hookrightarrow \mathcal{B}$.
If Φ is surjective (that is, for all $b \in B$ there exists an $a \in A$ such that $\Phi(a) = b$), then Φ is called epimorphism, indicated by $\Phi : \mathcal{A} \twoheadrightarrow \mathcal{B}$.
A homomorphism that is both injective and surjective (and thus is bijective) is called isomorhism, indicated by $\Phi : \mathcal{A} \hookrightarrow\!\!\!\!\rightarrow \mathcal{B}$.

Two Boolean algebras $\mathcal{A}$ and $\mathcal{B}$ are called isomorphic, noted by $\mathcal{A} \cong \mathcal{B}$ **iff** *there is an isomorphism $\Phi : \mathcal{A} \hookrightarrow\!\!\!\!\rightarrow \mathcal{B}$.*
The image of $\mathcal{A}$ in $\mathcal{B}$ under Φ, written $\Phi\mathcal{A}$, is defined as $\Phi\mathcal{A} =< \Phi(A), *_B, +_B, {}^-{}_B, 1_B, 0_B >$ where $\Phi(A) = \{b \in B \mid$ exists $a \in A$ such that $\Phi(a) = b\}$.

The following lemma shows that homomorphisms preserve the order of elements as well as the 0 and 1 elements. Furthermore, the image of a Boolean algebra is a subalgebra and monomorphisms map non-zero elements to non-zero elements.

Lemma 4.1.14
Let $\Phi : \mathcal{A} \to \mathcal{B}$ be a homomorphism. Then

1. *$a_1 \leq a_2$ implies $\Phi(a_1) \leq \Phi(a_2)$.*
   If Φ in addition is injective, then $a_1 \leq a_2$ **iff** *$\Phi(a_1) \leq \Phi(a_2)$.*
2. *$\Phi(0) = 0$ and $\Phi(1) = 1$*
3. *$\Phi\mathcal{A}$ is a subalgebra of $\mathcal{B}$*
4. *If Φ is injective, then $a \neq 0$ implies $\Phi(a) \neq 0$.*

Proof:

1. $\Phi(a_1) * \Phi(a_2) = \Phi(a_1 * a_2) \overset{a_1 * a_2 = a_1}{=\!=} \Phi(a_1)$.
   To prove $\Phi(a_1) \leq \Phi(a_2)$ implies $a_1 \leq a_2$, one has to show that $\Phi(a_1) * \Phi(a_2) = \Phi(a_1)$ implies $a_1 * a_2 = a_1$. Because $\Phi(a_1) * \Phi(a_2) = \Phi(a_1 * a_2)$, it is sufficient to show $\Phi(a_1 * a_2) = \Phi(a_1)$ implies $a_1 * a_2 = a_1$. But since Φ is injective, $\Phi(a_1 * a_2) = \Phi(a_1)$ implies $\Phi^{-1}\Phi(a_1 * a_2) = \Phi^{-1}\Phi(a_1)$ from which $a_1 * a_2 = a_1$ follows trivially.
2. $\Phi(0) = \Phi(0 * \overline{0}) = \Phi(0) * \Phi(\overline{0}) = \Phi(0) * \overline{\Phi(0)} = 0$
   $\Phi(1) = \Phi(\overline{0}) = \overline{\Phi(0)} = \overline{0} = 1$
3. By definition $\Phi(A) \subseteq B$.
   Suppose $b_1, b_2 \in \Phi(A)$. Then there are $a_i \in A$ such that $\Phi(a_i) = b_i$. Now:
   $b_1 * b_2 = \Phi(a_1) * \Phi(a_2) = \Phi(a_1 * a_2) \in \Phi(A)$
   $b_1 + b_2 = \Phi(a_1) + \Phi(a_2) = \Phi(a_1 + a_2) \in \Phi(A)$
   $\overline{b_1} = \overline{\Phi(a_1)} = \Phi(\overline{a_1}) \in \Phi(A)$
4. Suppose not, that is $\Phi(a) = 0 \overset{2.}{=\!=} \Phi(0)$. Then, since Φ is injective $a = 0$. Contradiction.

■

Theorem 4.1.15 (Characterization of Complete and Atomic Boolean Algebras)
$\mathcal{A}$ is a complete, atomic algebra **iff** *$\mathcal{A} \cong < \mathcal{P}(S(\mathcal{A})), \cap, \cup, {}^-, S(\mathcal{A}), \emptyset >$.*
The isomorphism $\Phi : \mathcal{A} \hookrightarrow \mathcal{P}(S(\mathcal{A}))$ is given by
$\Phi(a) = \{\, s \in S(\mathcal{A}) \mid s \leq a \,\}$. Especially, $\Phi(s) = \{\, s \,\}$ for atoms $s \in S(\mathcal{A})$.

Proof:([Hal74],p.70,theorem 5)

Assume $\mathcal{A}$ is complete and atomic and let $a_1, a_2 \in \mathcal{A}$. Then

- $\Phi(a_1 * a_2) = \{ s \in S(\mathcal{A}) \mid s \leq a_1 * a_2 \}$
  $= \{ s \in S(\mathcal{A}) \mid s \leq a_1 \text{ and } s \leq a_2 \}$
  $= \{ s \in S(\mathcal{A}) \mid s \leq a_1 \} \cap \{ s \in S(\mathcal{A}) \mid s \leq a_2 \}$
  $= \Phi(a_1) \cap \Phi(a_2)$

- $\Phi(a_1 + a_2) = \{ s \in S(\mathcal{A}) \mid s \leq a_1 + a_2 \}$
  $= \{ s \in S(\mathcal{A}) \mid s \leq a_1 \text{ or } s \leq a_2 \}$
  $= \{ s \in S(\mathcal{A}) \mid s \leq a_1 \} \cup \{ s \in S(\mathcal{A}) \mid s \leq a_2 \}$
  $= \Phi(a_1) \cup \Phi(a_2)$
  The second equality holds due to Lemma 4.1.5.

- $\Phi(\bar{a}) = \{ s \in S(\mathcal{A}) \mid s \leq \bar{a} \}$
  $= \{ s \in S(\mathcal{A}) \mid s * a = 0 \}$
  $= S(\mathcal{A}) - \{ s \in S(\mathcal{A}) \mid s * a \neq 0 \}$
  $= S(\mathcal{A}) - \{ s \in S(\mathcal{A}) \mid s \leq a \}$
  $= \overline{\Phi(a)}$.

Therefore, Φ is a homomorphism.

Further, Φ is surjective: Let X be an arbitrary subset of $S(\mathcal{A})$. Then $x = \bigvee_{s \in X} s$ exists (as $\mathcal{A}$ is complete) and $\Phi(x) = X$.

Finally, Φ is injective: If $\Phi(a_1) = \Phi(a_2)$ then $\bigvee_{s \in \Phi(a_1)} s = \bigvee_{s \in \Phi(a_2)} s$. But since $\mathcal{A}$ is atomic, every element is equal to the supremum of elements it dominates (see Lemma 4.1.10). Thus $a_1 = a_2$.

In the opposite direction, $< \mathcal{P}(S(\mathcal{A})), \cap, \cup, ^-, S(\mathcal{A}), \emptyset >$ and thus $\mathcal{A} = \Phi^{-1}(\mathcal{P}(S(\mathcal{A})))$ are complete and atomic.

■

In the sequel, only complete, atomic Boolean algebras will be considered. The isomorphism mentioned above justifies the further restriction to Boolean algebras of sort $\mathcal{B} = \mathcal{P}(X)$ for arbitrary sets X (recall $\mathcal{P}(X)$ abbreviates $< \mathcal{P}(X), \cap, \cup, ^-, X, \emptyset >$).

4.1.5 Tensor Products

This sections deals with the task of embedding "small" algebras as independent subalgebras of one "big" algebra, called the tensor (exterior) product (compare [Kur94]).

Definition 4.1.16 (Tensor Product)
Given two Boolean set algebras $\mathcal{P}(X)$ and $\mathcal{P}(Y)$, their tensor product $\mathcal{P}(X) \otimes \mathcal{P}(Y)$ is given by $\mathcal{P}(X \times Y)$, that is $< \{T \mid T \subseteq X \times Y\}, \cap, \cup, X \times Y, \emptyset >$. The atoms of $\mathcal{P}(X) \otimes \mathcal{P}(Y)$ are the sets containing exactly one pair $(x, y) \in X \times Y$.

Example: Let $X = \{x, z\}$ and $Y = \{y, z\}$.
Then $\mathcal{P}(X \times Y) = < \mathcal{P}(\{(x, y), (x, z), (z, y), (z, z)\}), \cap, \cup, X \times Y, \emptyset >$. An element of $\mathcal{P}(X \times Y)$ is, for example, $\{(x, y), (z, z)\}$. □

Next, canonical embeddings and projections are defined:

Definition 4.1.17 (Canonical Monomorphisms and Projections)
Let $\mathcal{P}(X)$ and $\mathcal{P}(Y)$ be Boolean algebras, and $X_1 \subseteq X$, $Y_1 \subseteq Y$ and $T \subseteq X \times Y$. Then the two canonical monomorphisms (also referred to as liftings) φ_X and φ_Y are defined by

$$\varphi_X : \begin{cases} \mathcal{P}(X) & \to \mathcal{P}(X) \otimes \mathcal{P}(Y) \\ X_1 & \mapsto \varphi_X(X_1) = \{(x_1, y) \mid x_1 \in X_1, y \in Y\} = X_1 \times Y \end{cases}$$

$$\varphi_Y : \begin{cases} \mathcal{P}(Y) & \to \mathcal{P}(X) \otimes \mathcal{P}(Y) \\ Y_1 & \mapsto \varphi_Y(Y_1) = \{(x, y_1) \mid x \in X, y_1 \in Y_1\} = X \times Y_1 \end{cases}$$

The canonical projections $\big|_X$ and $\big|_Y$ are defined by:

$$\big|_X : \begin{cases} \mathcal{P}(X) \otimes \mathcal{P}(Y) & \to \mathcal{P}(X) \\ T & \mapsto T\big|_X = \{x \in X \mid \exists_{y \in Y}\,(x, y) \in T\} \end{cases}$$

$$\big|_Y : \begin{cases} \mathcal{P}(X) \otimes \mathcal{P}(Y) & \to \mathcal{P}(Y) \\ T & \mapsto T\big|_Y = \{y \in Y \mid \exists_{x \in X}\,(x, y) \in T\} \end{cases}$$

Some special cases :

- $\varphi_X(X) = X \times Y$
- $\varphi_X(\{x\}) = \{x\} \times Y$ for $x \in X$
  (these are the atoms of the subalgebra $\varphi_X(\mathcal{P}(X))$ of $\mathcal{P}(X) \otimes \mathcal{P}(Y)$),
- $\varphi_X(\emptyset) = \emptyset$
- $\{(x, y)\}\big|_X = \{x\}$

Lemma 4.1.18

1. *φ_X and φ_Y are monomorphisms,*
2. *$\varphi_X(\mathcal{P}(X))$ and $\varphi_Y(\mathcal{P}(Y))$ are independent subalgebras of $\mathcal{P}(X) \otimes \mathcal{P}(Y)$,*
3. *$(\varphi_X(X_1))\big|_X = X_1$ and $(\varphi_Y(Y_1))\big|_Y = Y_1$,*
4. *$\varphi_X(T\big|_X) \subseteq T$ and $\varphi_Y(T\big|_Y) \subseteq T$,*
5. *$\big|_X$ and $\big|_Y$ are not homomorphisms,*
6. *$(T_1 \cap T_2)\big|_X \subseteq T_1\big|_X \cap T_2\big|_X$,*
7. *$T \neq \emptyset$ **iff** $T\big|_X \neq \emptyset$,*
8. *$\big|_X$ maps atoms $\{(x, y)\}$ of $\mathcal{P}(X) \otimes \mathcal{P}(Y)$ to atoms $\{x\}$ of $\mathcal{P}(X)$,*

9. $T_1 \subseteq T_2$ *implies* $T_1\big|_X \subseteq T_2\big|_X$.

Proof:

1. $\varphi_X(X_1) \cup \varphi_X(X_2) = (X_1 \times Y) \cup (X_2 \times Y) = (X_1 \cup X_2) \times Y = \varphi_X(X_1 \cup X_2)$
   $\varphi_X(X_1) \cap \varphi_X(X_2) = (X_1 \times Y) \cap (X_2 \times Y) = (X_1 \cap X_2) \times Y = \varphi_X(X_1 \cap X_2)$
   $\varphi_X(\overline{X_1}) = (X - X_1) \times Y = (X \times Y) - (X_1 \times Y) = (X \times Y) - \varphi_X(X_1) = \overline{\varphi_X(X_1)}$
   $\varphi_X(X_1) = \varphi_X(X_2)$ implies $(X_1 \times Y) = (X_2 \times Y)$ implies $X_1 = X_2$

2. $\varphi_X(\mathcal{P}(X)) =< \{X_1 \times Y \mid X_1 \subseteq X\}, \cap, \cup, X \times Y, \emptyset >$.
   Clearly, $\{X_1 \times Y \mid X_1 \subseteq X\} \subseteq \{T \mid T \subseteq X \times Y\}$ and
   $\varphi_X(X_1) \cup \varphi_X(X_2) = (X_1 \times Y) \cup (X_2 \times Y) = (X_1 \cup X_2) \times Y \in \varphi_X(\mathcal{P}(X))$,
   $\varphi_X(X_1) \cap \varphi_X(X_2) = (X_1 \times Y) \cap (X_2 \times Y) = (X_1 \cap X_2) \times Y \in \varphi_X(\mathcal{P}(X))$,
   $\overline{\varphi_X(X_1)} = (X \times Y) - \varphi_X(X_1) = (X \times Y) - (X_1 \times Y) = (X - X_1) \times Y \in \varphi_X(\mathcal{P}(X))$.
   It remains to show that $\varphi_X(\mathcal{P}(X))$ and $\varphi_Y(\mathcal{P}(Y))$ are independent. Suppose $\emptyset \neq \varphi_X(X_1) \in \varphi_X(\mathcal{P}(X))$ and $\emptyset \neq \varphi_Y(Y_1) \in \varphi_Y(\mathcal{P}(Y))$. That is, there are elements $x_1 \in X_1$ and $y_1 \in Y_1$ such that $(x_1, y) \in \varphi_X(X_1)$ for all $y \in Y$ and $(x, y_1) \in \varphi_Y(Y_1)$ for all $x \in X$. Thus $(x_1, y_1) \in \varphi_X(X_1) \cap \varphi_Y(Y_1) \neq \emptyset$.

3. $(X_1 \times Y)\big|_X = \{x \in X \mid \exists_{y \in Y}\, (x, y) \in (X_1 \times Y)\} = \{x \in X \mid x \in X_1\} = X_1$

4. Suppose $(x_0, y_0) \in T$.
   $\curvearrowright$ $x_0 \in T\big|_X$
   $\curvearrowright$ $(x_0, y_0) \in \varphi_X(T\big|_X)$

5. Take $x_1, x_2 \in X$ such that $x_1 \neq x_2$.
   Then $((\{x_1\} \times Y) \cap (\{x_2\} \times Y))\big|_Y = \emptyset\big|_Y = \emptyset \subsetneq Y = Y \cap Y = (\{x_1\} \times Y)\big|_Y \cap (\{x_2\} \times Y)\big|_Y$

6. $(T_1 \cap T_2)\big|_X = \{x \in X \mid \exists_{y \in Y}\, (x, y) \in T_1 \cap T_2\} \subseteq \{x \in X \mid \exists_{y_1 \in Y}\, (x, y_1) \in T_1 \text{ and } \exists_{y_2 \in Y}\, (x, y_2) \in T_2\} = \{x \in X \mid \exists_{y_1 \in Y}\, (x, y_1) \in T_1\} \cap \{x \in X \mid \exists_{y_2 \in Y}\, (x, y_2) \in T_2\} = T_1\big|_X \cap T_2\big|_X$

7. By definition.

8. By definition.

9. One has to show $T_1\big|_X \cap T_2\big|_X = T_1\big|_X$.
   Obviously, $T_1\big|_X \cap T_2\big|_X \subseteq T_1\big|_X$. It remains to show $T_1\big|_X \subseteq T_1\big|_X \cap T_2\big|_X$. But since $T_1 \cap T_2 = T_1$, this is equivalent to $(T_1 \cap T_2)\big|_X \subseteq T_1\big|_X \cap T_2\big|_X$ which was shown above.

■

The second property is called the lifting lemma in [Kur94].

REMARK 1. Tensor products should not be confused with the more familiar direct products defined as follows:
$\mathcal{A} \times \mathcal{B} =< A \times B, *, +, ^-, 1, 0 >$ with $(a_1, b_1) * (a_2, b_2) \stackrel{\text{def}}{=} (a_1 * a_2, b_1 * b_2)$, $(a_1, b_1) + (a_2, b_2) \stackrel{\text{def}}{=} (a_1 + a_2, b_1 + b_2)$, $\overline{(a, b)} \stackrel{\text{def}}{=} (\bar{a}, \bar{b})$, $1 \stackrel{\text{def}}{=} (1, 1)$, and $0 \stackrel{\text{def}}{=} (0, 0)$. One difference in the theory of direct products is that the projection functions $\big|_{\mathcal{A}}$ and $\big|_{\mathcal{B}}$ defined by $(a, b)\big|_{\mathcal{A}} = a$ and $(a, b)\big|_{\mathcal{B}} = b$ trivially are homomorphisms. Further, the number of elements in $\mathcal{A} \times \mathcal{B}$ is given by $2^{|A|+|B|}$ whereas $\mathcal{A} \otimes \mathcal{B}$ has $2^{|A|*|B|}$ elements.

REMARK 2. Because $\mathcal{P}(X) \otimes (\mathcal{P}(Y) \otimes \mathcal{P}(Z)) = \mathcal{P}(X \times Y \times Z) = (\mathcal{P}(X) \otimes \mathcal{P}(Y)) \otimes \mathcal{P}(Z)$, all definitions and lemmas about tensor products can easily be generalized to deal with a finite number of component algebras. For example, the tensor product of n Boolean algebras $\mathcal{P}(X_1), \ldots, \mathcal{P}(X_n)$ is defined by
$\mathcal{P}(X_1) \otimes \cdots \otimes \mathcal{P}(X_n) \stackrel{\text{def}}{=} \mathcal{P}(X_1 \times \cdots \times X_n)$.

4.1.6 An Algebra of Transitions

In this section, the Boolean algebras introduced above are related to the logic defined in Chapter 3.

As in Definition 3.2.5, consider a signature consisting of a set Vars of variables and a set Acts of actions, the latter modeling (synchronous/nonblocking) channels. Each variable and action is typed such that $\mathcal{D}(x)$ denotes the domain of values of the variable x and $\mathcal{D}(A)$ denotes the domain of values of the action A.

Given a set of variables Vars, a (concrete) state wrt Vars is an element of

$$\prod_{x \in \mathsf{Vars}} \mathcal{D}(x)$$

A concrete state ζ thus defines a valuation of the variables in Vars, that is

$$\mathsf{Vars} \ni x \stackrel{\zeta}{\longmapsto} \zeta(x) \in \mathcal{D}(x)$$

Similarly, restricting a valuation α to the actions in Acts results in a total function $\alpha : \mathsf{Acts} \to \mathcal{U} \cup \{\bot\}$ such that:

$$\mathsf{Acts} \ni A \stackrel{\alpha}{\longmapsto} \alpha(A) \in \mathcal{D}_\bot(A)$$

Now the concept of a (concrete) transition wrt (Vars,Acts) may be formalized. It is a pair of states together with a valuation of each action, that is, an element of:

$$\mathsf{Transitions} \stackrel{\text{def}}{=} \prod_{x \in \mathsf{Vars}} \mathcal{D}(x) \times \prod_{A \in \mathsf{Acts}} \mathcal{D}_\bot(A) \times \prod_{x' \in \mathsf{Vars}'} \mathcal{D}(x')$$

Example (traffic light):

To model traffic lights that allow pedestrians to safely cross a street, let pedestrians and cars be two variables with values $\mathcal{D}(\mathsf{pedestrians}) = \{\mathsf{green}, \mathsf{red}\}$ and $\mathcal{D}(\mathsf{cars}) = \{\mathsf{green}, \mathsf{amber}, \mathsf{red}\}$. Possible states are, for example, $(\mathsf{red}, \mathsf{amber})$ and $(\mathsf{green}, \mathsf{green})$. Altogether, there are $2 * 3 = 6$ states. Pedestrians may issue a request by pressing a button. This is modeled as an action that carries no information. Thus $\mathcal{D}(\mathsf{Request})$ has only one element, say $\mathcal{D}(\mathsf{Request}) = \{\surd\}$, and Request occurs **iff** $\alpha(\mathsf{Request}) = \surd$.

A possible transition of the traffic light system is, for example,

$\binom{\mathsf{red}}{\mathsf{green}} \xrightarrow{\surd} \binom{\mathsf{red}}{\mathsf{amber}}$, noted according to the schema

$$\begin{pmatrix}\zeta(\mathsf{pedestrians})\\ \zeta(\mathsf{cars})\end{pmatrix} \xrightarrow{\alpha(\mathsf{Request})} \begin{pmatrix}\zeta(\mathsf{pedestrians}')\\ \zeta(\mathsf{cars}')\end{pmatrix} \quad .$$

Altogether, there are $6 * 2 * 6 = 72$ possible transitions. □

In order to model situations in which several "concrete" transitions are possible, it is reasonable to consider arbitrary sets of transitions, or, expressed otherwise, arbitrary elements of $\mathcal{P}(\mathsf{Transitions})$. This Boolean algebra may be derived as the tensor product of Boolean algebras corresponding to the variables and actions as follows:

Given a variable x of domain $\mathcal{D}(x)$, $\mathcal{P}(\mathcal{D}(x))$ defines a Boolean algebra. Similarly, for an action A of domain $\mathcal{D}(A)$, $\mathcal{P}(\mathcal{D}_\perp(A))$ defines a Boolean algebra. Given a set of variables Vars and a set of actions Acts, these definitions extend as follows:

$$\begin{aligned}\mathcal{B}_{\mathsf{Vars}} &\stackrel{\text{def}}{=} \mathcal{P}\Big(\prod_{x\in\mathsf{Vars}} \mathcal{D}(x)\Big) &&= \bigotimes_{x\in\mathsf{Vars}} \mathcal{P}(\mathcal{D}(x))\\ \mathcal{B}_{\mathsf{Acts}} &\stackrel{\text{def}}{=} \mathcal{P}\Big(\prod_{A\in\mathsf{Acts}} (\mathcal{D}_\perp(A))\Big) &&= \bigotimes_{A\in\mathsf{Acts}} \mathcal{P}(\mathcal{D}_\perp(A))\end{aligned}$$

This way, forming tuples of variables corresponds with forming tensor products as shown for two variables x and y in the following diagram:

$$\begin{array}{ccc}\mathcal{D}(x)\ ,\ \mathcal{D}(y) & \xrightarrow{\textit{power set construction}} & \mathcal{P}(\mathcal{D}(x))\ ,\ \mathcal{P}(\mathcal{D}(y))\\ \Big\downarrow \textit{cross product} & & \Big\downarrow \textit{tensor product}\\ \mathcal{D}(x)\times\mathcal{D}(y) & \xrightarrow{\textit{power set construction}} & \mathcal{P}(\mathcal{D}(x))\otimes\mathcal{P}(\mathcal{D}(y)) \;=\; \mathcal{P}(\mathcal{D}(x)\times\mathcal{D}(y))\end{array}$$

Given a set of variables Vars and actions Acts, let

$$\begin{aligned}\mathcal{B}_{(\mathsf{Vars},\mathsf{Acts})} &\stackrel{\text{def}}{=} \mathcal{B}_{\mathsf{Vars}}\otimes\mathcal{B}_{\mathsf{Acts}}\otimes\mathcal{B}_{\mathsf{Vars}}\\ &= \mathcal{P}\Big(\prod_{x\in\mathsf{Vars}}\mathcal{D}(x)\times\prod_{A\in\mathsf{Acts}}\mathcal{D}_\perp(A)\times\prod_{x\in\mathsf{Vars}}\mathcal{D}(x)\Big)\end{aligned}$$

That is, $\mathcal{B}_{(\mathsf{Vars},\mathsf{Acts})} = \mathcal{P}(\mathsf{Transitions})$ is the power set of the set of concrete transitions w.r.t. (Vars,Acts).

Example (traffic light):

In the traffic light example, a transition from state $(\mathsf{red},\mathsf{amber})$ to state $(\mathsf{green},\mathsf{red})$ should be possible independently from the value of Request, that is, whether the pedestrian presses the button again or not. Thus it is reasonable to summarize the two transitions

$$\begin{pmatrix}\mathsf{red}\\ \mathsf{amber}\end{pmatrix} \xrightarrow{\checkmark} \begin{pmatrix}\mathsf{green}\\ \mathsf{red}\end{pmatrix} \quad\text{and}\quad \begin{pmatrix}\mathsf{red}\\ \mathsf{amber}\end{pmatrix} \xrightarrow{\perp} \begin{pmatrix}\mathsf{green}\\ \mathsf{red}\end{pmatrix} \quad .$$

The element of $\mathcal{B}_{(\{\mathsf{pedestrians},\mathsf{cars}\},\{\mathsf{Request}\})}$ that "summarizes" the two transitions simply is

$$\left\{\ \begin{pmatrix}\mathsf{red}\\ \mathsf{amber}\end{pmatrix} \xrightarrow{\checkmark} \begin{pmatrix}\mathsf{green}\\ \mathsf{red}\end{pmatrix}\ ,\ \begin{pmatrix}\mathsf{red}\\ \mathsf{amber}\end{pmatrix} \xrightarrow{\perp} \begin{pmatrix}\mathsf{green}\\ \mathsf{red}\end{pmatrix}\ \right\} \quad .$$

This set may be represented shortly as $\begin{pmatrix}\mathsf{red}\\ \mathsf{amber}\end{pmatrix} \xrightarrow{\{\surd,\perp\}} \begin{pmatrix}\mathsf{green}\\ \mathsf{red}\end{pmatrix}$ with the intended meaning

to abbreviate all transitions

(1) that start from state $(\mathsf{red}, \mathsf{amber})$, *and*
(2) such that a Request occurs or does not occur, *and*
(3) that end in state $(\mathsf{green}, \mathsf{red})$.

This set may also be described as

$$\{(\xi, \alpha, \xi') \in \mathsf{Transitions} \mid \begin{array}{c}\xi(\mathsf{pedestrians})=\mathsf{red} \text{ and } \xi(\mathsf{cars})=\mathsf{amber}\\ \alpha(\mathsf{Request})=\surd \text{ or } \alpha(\mathsf{Request})=\perp\\ \xi'(\mathsf{pedestrians})=\mathsf{green} \text{ and } \xi'(\mathsf{cars})=\mathsf{red}\end{array}\}.$$

However, only $2^6 * 2^2 * 2^6 = 2^{14}$ of the 2^{72} possible sets of transitions may be abbreviated according to the schema $\mathsf{set\ of\ states} \xrightarrow{\mathsf{set\ of\ valuations\ of\ the\ actions}} \mathsf{set\ of\ states}$. □

Note that the algebra $\mathcal{B}_{(\mathsf{Vars},\mathsf{Acts})}$ perfectly fits to canonical embeddings and projections: Adding a variable v means embedding $\mathcal{B}_{(\mathsf{Vars},\mathsf{Acts})}$ into the larger algebra $\mathcal{B}_{(\mathsf{Vars}\cup\{v\},\mathsf{Acts})}$ whereas removing a variable v means performing a canonical projection to $\mathcal{B}_{(\mathsf{Vars}-\{v\},\mathsf{Acts})}$. Obviously, this also holds for actions.

Example (traffic light): Assume, the traffic light specification has to be embedded into a larger system that also allows to count the number of requests. Then, the algebra for this larger system might be based on a set $\mathsf{Vars1} \stackrel{\text{def}}{=} \{\mathsf{pedestrians}, \mathsf{cars}, \mathsf{count}\}$.

A transition is, for example, $\begin{pmatrix}\mathsf{red}\\ \mathsf{amber}\\ 7\end{pmatrix} \xrightarrow{\surd} \begin{pmatrix}\mathsf{green}\\ \mathsf{red}\\ 8\end{pmatrix}$.

Similarly, if the lights for the cars were considered irrelevant, then the algebra may be projected canonically to $\mathcal{B}_{(\{pedestrians\},\{Request\})}$.

Then, the transition from above simplifies to $(\mathsf{red}) \xrightarrow{\surd} (\mathsf{green})$. □

4.2 Boolean Transition Systems

This section introduces Boolean transition systems (or BTS for short) as a means of representing TLT specifications as automata. Boolean transition systems are closely related to the ω-automata of [Kur94] and to the Boolean transition systems defined in [CGS91]. Labeling with elements of some Boolean algebra forms the common ground of these approaches.[2] As acceptance conditions, Büchi conditions as well as weak and strong fairness are considered. Büchi conditions are widely used in the field of ω-automata because they are compositional in the sense that the composition of two BTS with Büchi conditions can be represented again as BTS with Büchi conditions such that the (observable) behavior of the composed BTS is identical to the

[2] In difference to labeled transition systems whose labels are typically taken from some "flat" signature.

intersection of the behavior of the components. The drawback of Büchi conditions is however, that one can not guarantee in general that specifications with Büchi conditions are executable, i.e. that there is at least one behavior fulfilling the Büchi conditions. On the other hand, using only fairness conditions in fact guarantees that there are behavior of the BTS, however there is no smooth notion of composition any more: depending on the concrete definition of composition, one either looses the theorem that the behavior of the composition is identical to the intersection of the behavior of the components, or the existence of behavior can not be guaranteed for the composition.

To overcome these restrictions, I proceed as follows: Firstly, all acceptance conditions are expressed as Büchi conditions which guarantees the composition theorem mentioned above. For weak fairness conditions this is trivial. Strong fairness conditions however can not be expressed directly as Büchi conditions. Therefore they get replaced by weak fairness conditions at the cost of doubling the state space.

In a second step I focus on executable specifications by allowing only fairness conditions. This, in addition with some composition criteria is sufficient for guaranteeing that the composition of executable specifications is again executable. These criteria are weaker than the notion of being *input enabled* as found for example in [LT87, LT89] in the context of I/O-automata.

4.2.1 Boolean Transition Systems, Runs and Traces

This section summarizes the elementary definitions regarding Boolean transition systems.

Definition 4.2.1 (Boolean Transition Systems)
A Boolean transition system (BTS) is a structure $\Sigma = (V, I, \mathcal{B}, st, M, \mathcal{F})$ *such that:*

- V *is a set of states (vertices).*
- I *is a non-empty set of initial states, that is* [Init] $\emptyset \neq I \subseteq V$.
- $\mathcal{B}$ *is a complete, atomic Boolean algebra.*
- M *is a transition matrix, that is,* $M : V \times V \to \mathcal{B}$.
  $e \in V \times V$ *is called an edge* **iff** $M(e) \neq 0$.
- st *is a (non-zero) element of* $\mathcal{B}$*, such that* [St] $0 < st * M(v, v)$ *for all* $v \in V$.
  An atom $s \leq st$ *is called stutter step.*
- $\mathcal{F}$ *is a finite set of acceptance conditions. Each acceptance condition is either*
  (1) a weak fairness condition $\mathsf{WF}(E, L)$,
  (2) a strong fairness condition $\mathsf{SF}(E, L)$ *or*
  (3) a Büchi condition $\mathsf{Büchi}(E, L)$
  where E *is a set of edges and* L *is a function* $L : E \to \mathcal{B}$ *such that*

  [Fair] $0 < L(e) * M(e)$ *for all* $e \in E$

  Further, let G *be the projection of* $E \subseteq V \times V$ *to its first component, that is,* $G \stackrel{def}{=} \{v \in V \mid$ *exists* $w \in V$ *such that* $(v, w) \in E\}$. *Sometimes, it is convenient to denote* $\mathsf{WF}(E, L)$ *as* $\mathsf{WF}(G, E, L)$ *and* $\mathsf{SF}(E, L)$ *as* $\mathsf{SF}(G, E, L)$.

From [St], it follows immediately that

Corollary 4.2.2 *$M(v,v) \neq 0$ for all $v \in V$, that is, (v,v) is an edge for arbitrary $v \in V$.*

REMARK 1. *st* plays a crucial role to ensure the existence of traces and to guarantee compositionality. [St] allows a BTS to stutter in each state. However, stutter steps are not restricted to self loops as might be indicated by [St]. Indeed, stutter steps are concerned with the labeling of a BTS rather than with its states. In a sense to be made precise in Chapter 5, a BTS does some stutter step if only it does not force itself to take part in a synchronization or communication.

REMARK 2. [St] is a stronger requirement than the more common "$M(v,v) \neq 0$ for all $v \in V$."

REMARK 3. As $\mathcal{B}$ is atomic, $M(e) = \sum_{\substack{s \in S(\mathcal{B}) \\ s \leq M(e)}} s$ for all edges e. Thus an edge may be considered as being labeled by a set of atoms of the labeling Boolean algebra. These atoms will turn out to be valuations of variables and actions.

Next, the "visible behavior" of BTS is defined. This is done in two steps: in a first step, ignoring the labels, the set of runs through the BTS gets defined. On the basis of runs, traces get defined in a second step. Our interest lies mainly in these traces. Runs just are an aid in defining the set of traces. In addition, some proofs of theorems about traces can be carried out by simply considering runs.

A run is a path through the BTS that starts in an initial state, follows edges and respects the part of the acceptance conditions dealing with the edges.

Definition 4.2.3 (Runs)
A sequence $(v_0, v_1, \dots, v_n, \dots) \in V^\omega$ is called a run *of a BTS $\Sigma = (V, I, \mathcal{B}, st, M, \mathcal{F})$ iff*

[Run1] $v_0 \in I$,

[Run2] $M(v_i, v_{i+1}) \neq 0$ *for all* $i \in \mathbb{N}$,

[Run3] *for all* $\mathsf{WF}(G,E,L) \in \mathcal{F}$: *Infinitely often* $v_i \notin G$ *or infinitely often* $(v_i, v_{i+1}) \in E$,
for all $\mathsf{SF}(G,E,L) \in \mathcal{F}$: *Infinitely often* $v_i \in G$ *implies infinitely often* $(v_i, v_{i+1}) \in E$,
and
for all $\mathsf{Büchi}(E,L) \in \mathcal{F}$: *Infinitely often* $(v_i, v_{i+1}) \in E$.

Runs(Σ) *is the set of all runs of* Σ. *A (finite) sequence* $(v_0, v_1, \dots v_n) \in V^n$ *is called initial run,* **iff** *it fulfills* $v_0 \in I$ *and* $M(v_i, v_{i+1}) \neq 0$ *for all* $0 \leq i < n$.

Mostly, weak fairness will be used as acceptance condition. Two equivalent characterizations for weak fairness are

Corollary 4.2.4

A sequence $(v_0, v_1, \dots, v_n, \dots) \in V^\omega$ satisfies $\mathsf{WF}(G,E,L)$

iff *If from some point k on $v_i \in G$ for all $i \geq k$ then there is a $l \geq k$ such that $(v_l, v_{l+1}) \in E$*

iff *It is not the case that there is a (fixed but arbitrary) k such that $v_i \in G$ for all $i \geq k$ and $(v_i, v_{i+1}) \notin E$ for all $i \geq k$.*

Definition 4.2.5 (Traces)
A sequence $(s_0, s_1, \dots)$ of atoms of $\mathcal{B}$ is a trace of Σ **iff** *Σ has a run $(v_0, \dots)$ such that*

[Tr1] $s_i \leq M(v_i, v_{i+1})$.

[Tr2] *for all* $\mathsf{WF}(G, E, L) \in \mathcal{F}$ *: Infinitely often* $v_i \notin G$ *or infinitely often* $s_i \leq L(v_i, v_{i+1})$ *and* $(v_i, v_{i+1}) \in E$,
for all $\mathsf{SF}(G, E, L) \in \mathcal{F}$ *: Infinitely often* $v_i \in G$ *implies infinitely often* $s_i \leq L(v_i, v_{i+1})$ *and* $(v_i, v_{i+1}) \in E$, *and*
for all $\mathsf{Büchi}(E, L) \in \mathcal{F}$ *: Infinitely often* $s_i \leq L(v_i, v_{i+1})$ *and* $(v_i, v_{i+1}) \in E$.

The set of all traces of Σ *is referred to as* $\mathbf{Traces}(\Sigma)$.

Let $\mathcal{B}_L$ *be a (complete and atomic) Boolean algebra and* $\big|_{\mathcal{B}_L} : \mathcal{B} \twoheadrightarrow \mathcal{B}_L$ *the canonical projection. Then* $\mathbf{Traces}(\Sigma)\big|_{\mathcal{B}_L}$ *is defined as the set of all sequences* $(s_0, s_1, \dots) \in S(\mathcal{B}_L)^\omega$ *such that there exists a sequence* $(t_0, t_1, \dots) \in \mathbf{Traces}(\Sigma)$ *with* $t_0\big|_{\mathcal{B}_L} = s_0$.

Σ *is called executable* **iff** *there is at least one trace, that is* **iff** $\mathbf{Traces}(\Sigma) \neq \emptyset$.

From the definitions it follows that each run "dominates" at least one trace: if $(v_i, v_{i+1}) \notin E$ for any $F(E, L) \in \mathcal{F}$ then one chooses some arbitrary $s_i \leq M(v_i, v_{i+1})$ else one chooses some $F(E, L) \in \mathcal{F}$ such that $(v_i, v_{i+1}) \in E$ and $|\{j \mid (v_j, v_{j+1}) \in E \text{ and } j < i \text{ and } s_j \leq L(v_j, v_{j+1})\}|$ is maximal. Then one chooses some arbitrary $s_i \leq L(v_i, v_{i+1}) * M(v_i, v_{i+1})$. Due to [Fair], such atoms s_i always exist. Thus

Corollary 4.2.6

- *If* $\mathbf{Runs}(\Sigma) \neq \emptyset$ *then* $\mathbf{Traces}(\Sigma) \neq \emptyset$. *Further,*
- *a trace fulfilling* $\mathsf{SF}(E, L)$ *also fulfills* $\mathsf{WF}(E, L)$ *and*
- *a trace fulfills* $\mathsf{WF}(G, E, L)$ **iff** *it fulfills* $\mathsf{Büchi}(\hat{E}, \hat{L})$ *where*
  $\hat{E} = \{(v_1, v_2) \in V^2 \mid v_1 \notin G \text{ or } (v_1, v_2) \in E\}$ *and*
  $$\hat{L}(v_1, v_2) = \begin{cases} L(v_1, v_2) & \text{, for } (v_1, v_2) \in E \\ M(v_1, v_2) & \text{, for } (v_1, v_2) \notin E \end{cases}$$

Thus to guarantee that Σ is executable, it is sufficient to guarantee the existence of runs.

In absence of acceptance conditions, $(v, v, \dots)$ is a run for arbitrary $v \in I$ and $(s, s, \dots)$ is a trace dominated by $(v, v, \dots)$ for any atom $s \leq st * M(v, v)$. As a consequence, all BTS without acceptance conditions are trivially executable. The other way round, the only way to exclude the possibility of stuttering forever from the set of traces, or, equivalently, to guarantee any progress, is by means of acceptance conditions.

When TLT specifications are to be implemented in an operational manner (e.g. by compiling them into deterministic executable programs), one has to guarantee that any chosen initial trace can be completed to one satisfying the acceptance conditions. For arbitrary BTS this can not be guaranteed. However, problems only arise due to Büchi-acceptance conditions (as sketched in Figure 4.2). For the subclass of BTS without Büchi-acceptance conditions, this property will be proven in section 4.2.5

4.2.2 A Graphical Notation for Boolean Transition Systems

Pictures often provide some help in understanding particular aspects of a system description. But, as Leslie Lamport points out in [Lam94a], "for a picture to provide more than an informal

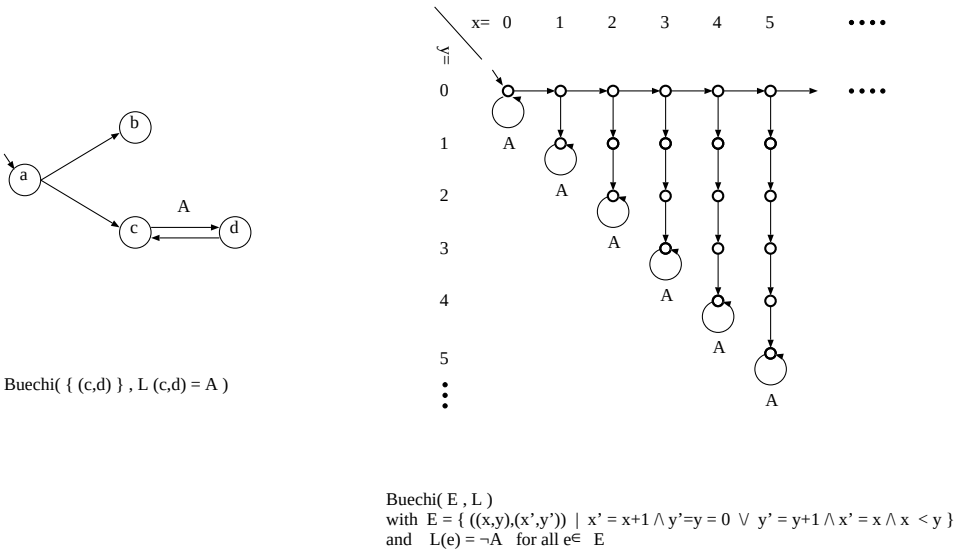


Figure 4.2: In the BTS shown on the left, the Büchi condition can be satisfied, but only if the step leaving state a leads to state c and not to b. However, one could argue that this Büchi condition still can be implemented, because to take that decision, it is sufficient to "look ahead" one transition. The Büchi condition in the BTS shown on the right states that traces may not end up in one of the self-loops labeled with A. Obviously, for this example it is not possible with any finite look ahead to determine a proper run (in position $(x = n, y = 0)$ a look-ahead of n transitions is necessary to decide not to increase y).

comment, there must be a formal connection between the complete specification and the picture". As a consequence, Lamport defines the semantics (the meaning) for diagrams to be a TLA formula Δ, and calls a diagram to be a diagram *for* a TLA specification Π **iff** $\Pi \Rightarrow \Delta$ holds. This definition allows

(1) incomplete diagrams that only describe certain aspects of a specification, as well as
(2) different diagrams that provide complementary views of one sole specification.

Especially, Lamport does not try to visualize fairness aspects in his diagrams. That is, he does not try to describe the complete temporal behavior of a specification in his diagrams.

The view taken in this thesis differs from Lamport's one in that the graph of a BTS as defined below fully captures the semantics of the BTS. More formally, there is a bijection between Boolean transition systems and their graphs, that is, each graph Δ corresponds uniquely to one BTS Σ. Due to this bijection each graph Δ defines a set of traces, namely $\mathbf{Traces}(\Delta) \stackrel{\text{def}}{=} \mathbf{Traces}(\Sigma)$ for the corresponding BTS Σ. With this in mind, it makes sense to talk of complementary abstract graphs of one BTS: if $\mathbf{Traces}(\Sigma_1) \subseteq \mathbf{Traces}(\Delta_2)$ then the graph Δ_2 corresponding to a BTS Σ_2 is called an *abstract* graph for Σ_1. For example, omitting the weak fairness conditions in a graph corresponding to a BTS Σ results in an abstract graph for Σ. Thus, neither (1) nor (2) of Lamport's requirements are lost, and in addition, a direct correspondence between BTS and their graphs is gained.

Given a Boolean algebra $\mathcal{B}$ and a stutter element st, the graph corresponding to a BTS $\Sigma = (V, I, \mathcal{B}, st, M, \mathcal{F})$ is a directed graph with nodes V and initial nodes I with the edges (i.e. any $e \in V \times V$ with $M(e) \neq 0$) being labeled by $M(e)$ as well as by a set of tuples $(color(\mathsf{F}(E, L)), L(e))$ for any acceptance condition $\mathsf{F}(E, L)) \in \mathcal{F}$ such that $e \in E$. Here $color : \mathcal{F} \to Id$ is an injective

function mapping $\mathcal{F}$ to an arbitrary set Id of identifiers. Given $\mathcal{B}$ and st, it is straightforward to regain V, I and M from a graph; an element $\mathsf{F}(E,L) \in \mathcal{F}$ is restored by collecting all edges labeled with one color.

Edges that are not affected by any acceptance condition, that is edges e such that $e \notin E$ for any $\mathsf{F}(E,L)) \in \mathcal{F}$, are called non-colored.

To reduce the labeling of most graphs, a small set of conventions suffices:

1. Non-colored edges are printed as dashed arrows and labeled by $M(e)$.
2. Non-colored self loops labeled with a label l such that $l \leq st$ are omitted.
3. If $L(e)$ agrees with $M(e)$, then the label $(color(\mathsf{F}(E,L)), L(e))$ is simplified to $color(\mathsf{F}(E,L))$.
4. In case a BTS has only one sole acceptance condition $\mathsf{F}(E,L)$, the labels $(color(\mathsf{F}(E,L)), L(e))$ are omitted completely whenever $L(e) = M(e)$.

Sometimes, the graphs are also simplified by explicitly noting the acceptance conditions below the graph instead of overloading the graph with the "colors".

Besides these general rules, graphs can be further adjusted to the common situation (compare Section 4.1.6) where $\mathcal{B} = \mathcal{B}_{(\mathsf{Vars},\mathsf{Acts})} = \mathcal{P}(\prod_{x \in \mathsf{Vars}} \mathcal{D}(x) \times \prod_{A \in \mathsf{Acts}} \mathcal{D}_\perp(A) \times \prod_{x' \in \mathsf{Vars}'} \mathcal{D}(x'))$ and $V = \prod_{x \in \mathsf{Vars}} \mathcal{D}(x)$:

Firstly, a predicate p may be used to label a state v if v is the sole valuation satisfying p.

Secondly, if (v_1, v_2) is an edge and the v_i are labeled with p_i, then the edge may be labeled with p instead of $M(v_1, v_2)$ provided that

$$[\![p_1 \ \wedge \ p \ \wedge \bigwedge_{\substack{A \in \mathsf{Acts} \\ A \text{ does not occur in } p}} \neg A \ \wedge \ p_2']\!]^\beta \quad \textbf{iff} \quad \beta \in M(e) \ .$$

The label true may be completely omitted. Note, that due to the last rule a label true implies that no action occurs.

Thirdly, for an action A, $A(c)?$ replaces the label $(A(c) \vee \neg A)$.

Example

Consider the BTS $\Sigma(\text{example}) = (V, I, \mathcal{B}, st, M, \mathcal{F})$ being defined by

- $V = \{\mathsf{true}, \mathsf{false}\}$,
- $I = \{\mathsf{true}\}$,
- $\mathcal{B} = \mathcal{B}_{(\{b\},\{A,B\})}$ for a Boolean valued variable b and two signals A and B ,
- $M(\mathsf{true}, \mathsf{true}) = \{(\mathsf{true}, \perp, \perp, \mathsf{true})\}$
  $M(\mathsf{true}, \mathsf{false}) = \{(\mathsf{true}, \surd, \perp, \mathsf{false})\}$
  $M(\mathsf{false}, \mathsf{true}) = \emptyset$
  $M(\mathsf{false}, \mathsf{false}) = \{(\mathsf{false}, \perp, \perp, \mathsf{false}), (\mathsf{false}, \perp, \surd, \mathsf{false})\}$,

- $st = \{(\mathsf{true}, \bot, \bot, \mathsf{true}), (\mathsf{false}, \bot, \bot, \mathsf{false})\}$ and
- $\mathcal{F} = \emptyset$.

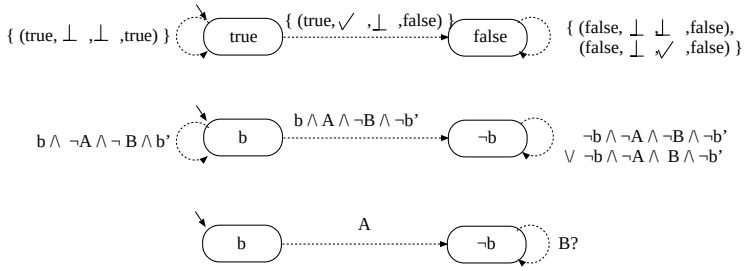


Figure 4.3: From top to bottom, the "original" graph for the BTS Σ(example), the graph with "complete" predicates as labels, and the graph with "short" predicates are shown. The elements of $\mathcal{B}_{(\{b\},\{A,B\})}$ are denoted as $(\zeta(b), \alpha(A), \alpha(B), \zeta'(b'))$.

Figure 4.3 shows the graphical representation of the BTS. □

4.2.3 Elimination of strong fairness

Strong fairness conditions can not be expressed directly as Büchi conditions. Instead there is a straightforward translation into (more general) Street conditions[3]. As a result, specifications using strong fairness conditions turned out to result in significantly higher computational complexities[4] than comparable specifications with weak fairness conditions. Nevertheless both weak and strong fairness are expressively equivalent. This observation was the origin of the following construction that replaces and under reasonable preconditions implements strong fairness by weak fairness at the cost of only one extra bit. In the context of this thesis, the construction helps simplifying the presentation of composition and executable BTS, because attention may be restricted to weak fairness and Büchi conditions.

To implement the strong fairness $\mathsf{SF}(G, E, L)$ by weak fairness conditions, the state space of the original BTS is doubled. The key idea is to define a second BTS, identical to the original one besides the fact that the only edges leaving G are elements of E. Thus if a path enters G in this copy then executing an edge from E is inevitable if only E is weak fair. To forward from the original BTS to its copy, all edges of the original BTS are doubled. That is, edges $(v_0, \tilde{v_1})$ are added for all original edges (v_0, v_1) where $\tilde{v_1}$ denotes the vertex of the copy corresponding to v_1. What remains is changing always eventually to the copy. This is achieved by requiring weak fairness for the edges $(v_0, \tilde{v_1})$. The construction is shown for an example in Figure 4.4.

Theorem 4.2.7
Let $\Sigma = (V, I, \mathcal{B}, st, M, \mathcal{F})$ be a BTS such that $\mathsf{SF}(E_0, L_0) \in \mathcal{F}$.

Then $\mathbf{Traces}(\Sigma) \subseteq \mathbf{Traces}(\tilde{\Sigma})$ *for* $\tilde{\Sigma} = (\tilde{V}, \tilde{I}, \mathcal{B}, st, \tilde{M}, \tilde{\mathcal{F}})$ *where*

$\tilde{V} = V \times \{0, 1\}$,

[3] $\mathsf{Street}(E_1, L_1, E_2, L_2)$ holds for a trace $(s_0, s_1, \dots)$ **iff** infinitely often $e_i \in E_1$ and $s_i \leq L_1(e_i)$ implies infinitely often $e_i \in E_2$ and $s_i \leq L_2(e_i)$.

[4] In experiments carried out with the model checker SVE

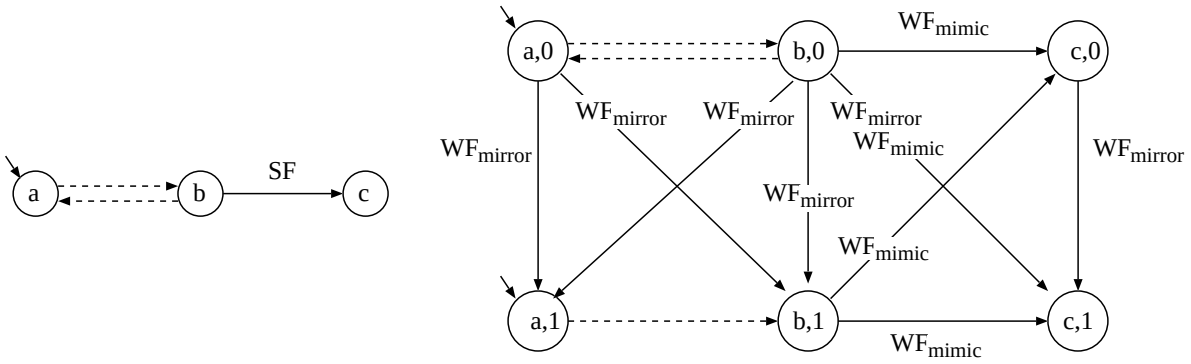


Figure 4.4: The strong fairness condition of the left BTS is "implemented" on the right.

$$\tilde{I} = I \times \{0,1\},$$

$$\tilde{M}((v_1,0),(v_2,d)) = M(v_1,v_2) \qquad \text{for } d \in \{0,1\}$$

$$\tilde{M}((v_1,1),(v_2,1)) = \begin{cases} M(v_1,v_2) & , v_1 \notin G_0 \text{ or } v_1 = v_2 \text{ or } (v_1,v_2) \in E_0 \\ 0 & , v_1 \in G_0 \text{ and } v_1 \neq v_2 \text{ and } (v_1,v_2) \notin E_0 \end{cases}$$

$$\tilde{M}((v_1,1),(v_2,0)) = \begin{cases} M(v_1,v_2) & , (v_1,v_2) \in E_0 \\ 0 & , \text{ else} \end{cases}$$

$$\tilde{\mathcal{F}} = \bigcup_{\mathsf{F}(E,L) \in \mathcal{F} - \{\mathsf{SF}(E_0,L_0)\}} \mathsf{F}(\tilde{E},\tilde{L}) \;\cup\; \mathsf{WF}_{mirror} \;\cup\; \mathsf{WF}_{mimic}$$

where $\mathsf{F}(\tilde{E},\tilde{L})$ *(for* $\mathsf{F} \in \{\mathsf{WF},\mathsf{SF},\mathsf{Büchi}\}$*) is defined by*

$$\tilde{E} \stackrel{def}{=} \{\; ((v_1,d_1),(v_2,d_2)) \in (V \times \{0,1\})^2 \mid (v_1,v_2) \in E \text{ and } \tilde{M}((v_1,d_1),(v_2,d_2)) \neq 0 \;\} \quad ,$$

$$\tilde{L}((v_1,d_1),(v_2,d_2)) \stackrel{def}{=} L(v_1,v_2) \quad \text{for all } ((v_1,d_1),(v_2,d_2)) \in \tilde{E} \quad ,$$

$$\mathsf{WF}_{mirror} \stackrel{def}{=} \mathsf{WF}(\;\{\; ((v_1,0),(v_2,1)) \in (V \times \{0,1\})^2 \mid M(v_1,v_2) \neq 0 \;\}\;,\; M\;) \quad , \text{ and}$$

$$\mathsf{WF}_{mimic} \stackrel{def}{=} \mathsf{WF}(\;\{\; ((v_1,d_1),(v_2,d_2)) \in (V \times \{0,1\})^2 \mid (v_1,v_2) \in E_0 \;\}\;,\; L_0\;) \;.$$

If furthermore for all fairness conditions $\mathsf{F}(G,E,L)$ *either*

$$G \cap G_0 = \emptyset$$

or for all $v_1 \in G \cap G_0$ *it holds that*

$$(v_1,v_2) \in E_0 \text{ implies } (v_1,v_2) \in E \text{ and } L_0(v_1,v_2) \leq L(v_1,v_2)$$

and if $\tilde{M}$ *gets redefined to*

$$\tilde{M}((v_1,0),(v_2,d)) = M(v_1,v_2) \qquad \text{for } d \in \{0,1\}$$

$$\tilde{M}((v_1,1),(v_2,1)) = \begin{cases} M(v_1,v_2) & , v_1 \notin G_0 \text{ or } v_1 = v_2 \text{ and } (v_1,v_2) \notin E_0 \\ M(v_1,v_2) * L_0(v_1,v_2) & , (v_1,v_2) \in E_0 \\ 0 & , v_1 \in G_0 \text{ and } v_1 \neq v_2 \text{ and } (v_1,v_2) \notin E_0 \end{cases}$$

$$\tilde{M}((v_1,1),(v_2,0)) = \begin{cases} M(v_1,v_2) * L_0(v_1,v_2) & , (v_1,v_2) \in E_0 \\ 0 & , \text{ else} \end{cases}$$

then $\mathbf{Traces}(\Sigma) = \mathbf{Traces}(\tilde{\Sigma})$ *and* $\tilde{\Sigma}$ *is called to implement* Σ.

Proof:

First of all, $\tilde{\Sigma}$ is well-defined: [Init] and [St] follow immediately from the definition. It remains to show [Fair] for all acceptance conditions in $\tilde{\mathcal{F}}$, that is $0 < \tilde{L}(((v_1,d_1),(v_2,d_2))) * \tilde{M}(((v_1,d_1),(v_2,d_2)))$ for all $((v_1,d_1),(v_2,d_2)) \in \tilde{E}$. This holds trivially for WF_{mirror} because $M_{mirror}(((v_1,d_1),(v_2,d_2))) = L_{mirror}(((v_1,d_1),(v_2,d_2))) = M(v_1,v_2)$ and for WF_{mimic} because $(v_1,v_2) \in E_0$ and $L_{mimic}(((v_1,d_1),(v_2,d_2))) = L_0(v_1,v_2)$. For any other acceptance condition $\mathsf{F}(\tilde{G},\tilde{E},\tilde{L}) \in \tilde{\mathcal{F}}$, taking the original definition of $\tilde{M}$, the proof obligation $0 < \tilde{L}(((v_1,d_1),(v_2,d_2))) * \tilde{M}(((v_1,d_1),(v_2,d_2)))$ is equivalent with $0 < L(v_1,v_2) * M(v_1,v_2)$ which is [Fair] for the original BTS Σ. For the redefined version of $\tilde{M}$, either $G \cap G_0 = \emptyset$ and thus the redefinition does not affect $\tilde{E}$ or the proof obligation $0 < \tilde{L}(((v_1,d_1),(v_2,d_2))) * \tilde{M}(((v_1,d_1),(v_2,d_2)))$ is equivalent with $0 < L(v_1,v_2) * M(v_1,v_2) * L_0(v_1,v_2) = M(v_1,v_2) * L_0(v_1,v_2)$ which is [Fair] for $\mathsf{SF}(E_0,L_0)$.

Let $(s_0,s_1,\dots) \in S(\mathcal{B})^\omega$.

Summarizing the definitions for runs and traces,

$(s_0,s_1,\dots) \in \mathbf{Traces}(\Sigma)$ **iff** there is a sequences $(v_0,v_1,\dots) \in V^\omega$ such that

[A1] $v_0 \in I$

[A2] $s_i \leq M(v_i,v_{i+1})$

[A3] for all $\mathsf{WF}(G,E,L) \in \mathcal{F}$: Infinitely often $v_i \notin G$ or infinitely often $s_i \leq L(v_i,v_{i+1})$ and $(v_i,v_{i+1}) \in E$,
for all $\mathsf{SF}(G,E,L) \in \mathcal{F}$: Infinitely often $v_i \in G$ implies infinitely often $s_i \leq L(v_i,v_{i+1})$ and $(v_i,v_{i+1}) \in E$, and
for all $\mathsf{Büchi}(E,L) \in \mathcal{F}$: Infinitely often $s_i \leq L(v_i,v_{i+1})$ and $(v_i,v_{i+1}) \in E$.

$(s_0,s_1,\dots) \in \mathbf{Traces}(\tilde{\Sigma})$ **iff** there is a sequence $((w_0,d_0),(w_1,d_1),\dots) \in (V \times \{0,1\})^\omega$ such that

[B1] $(w_0,d_0) \in I_1 \times \{0,1\}$

[B2] $s_i \leq \tilde{M}((w_i,d_i),(w_{i+1},d_{i+1}))$

[B3] for all $\mathsf{WF}(\tilde{G},\tilde{E},\tilde{L}) \in \tilde{\mathcal{F}}$: Infinitely often $(w_i,d_i) \notin \tilde{G}$ or infinitely often $s_i \leq \tilde{L}((w_i,d_i),(w_{i+1},d_{i+1}))$ and $((w_i,d_i),(w_{i+1},d_{i+1})) \in \tilde{E}$,
for all $\mathsf{SF}(\tilde{G},\tilde{E},\tilde{L}) \in \tilde{\mathcal{F}}$: Infinitely often $(w_i,d_i) \in \tilde{G}$ implies infinitely often $s_i \leq \tilde{L}((w_i,d_i),(w_{i+1},d_{i+1}))$ and $((w_i,d_i),(w_{i+1},d_{i+1})) \in \tilde{E}$, and
for all $\mathsf{Büchi}(\tilde{E},\tilde{L}) \in \tilde{\mathcal{F}}$: Infinitely often $s_i \leq \tilde{L}((w_i,d_i),(w_{i+1},d_{i+1}))$ and $((w_i,d_i),(w_{i+1},d_{i+1})) \in \tilde{E}$.

Traces$(\Sigma) \subseteq$ **Traces**$(\tilde{\Sigma})$:

To show the inclusion from left to right, one has to determine the d_i to define a run of the extended BTS $\tilde{\Sigma}$. In order to guarantee the additional weak fairness condition $\mathsf{WF}_{\mathsf{mirror}}$(requiring that always eventually the "mirror" gets visited[5]), the following strategy is applied in determing the run of the extended BTS $\tilde{\Sigma}$: The transition to the "mirror" takes place as soon as the original run "decides" that it is going to take an edge from E_0 the next time it enters a state in G_0.

First, two auxiliary functions are defined. $\mathsf{Enabled}(i)$ is the next position starting at i where it is possible to take an edge from E_0. $\mathsf{Executed}(i)$ is the next position starting at i where an edge from E_0 is taken. Both functions return ω if those positions do not exist. Obviously, $\mathsf{Enabled}(i) \leq \mathsf{Executed}(i)$ (as usual, $k < \omega$ and $\omega = \omega$). More formally,

$$\mathsf{Enabled}(i) = \begin{cases} \min\{k \mid k \geq i \text{ and } v_k \in G_0\} & \text{, there is some } k \geq i \text{ with } v_k \in G_0 \\ \omega & \text{, else} \end{cases}$$

$$\mathsf{Executed}(i) = \begin{cases} \min\{k \mid k \geq i \text{ and } (v_k, v_{k+1}) \in E_0\} & \text{, there is some } k \geq i \\ & \quad \text{with } (v_k, v_{k+1}) \in E_0 \text{ and } s_k \leq L_0(v_k, v_{k+1}) \\ \omega & \text{, else} \end{cases}$$

Then let $w_i = v_i$

$$\text{and} \quad d_i = \begin{cases} 0 & \text{, if } \mathsf{Enabled}(i) < \mathsf{Executed}(i) \\ 1 & \text{, else, i.e. if } \mathsf{Enabled}(i) = \mathsf{Executed}(i) \end{cases} .$$

[A1] **implies** [B1] trivial

[A2] **implies** [B2]

Firstly, note that from the original definition of $\tilde{M}$ it follows that either $\tilde{M}((v_i, d_i), (v_{i+1}, d_{i+1})) = M(v_i, v_{i+1})$ or $\tilde{M}((v_i, d_i), (v_{i+1}, d_{i+1})) = 0$. We are finished if the second situation does not occur for the strategy chosen to determine w_i and d_i. Two cases are to be distinguished:

(1) $d_i = d_{i+1} = 1$, $v_i \in G_0$, $(v_i, v_{i+1}) \notin E_0$.

$$\mathsf{Executed}(i) \overset{d_i=1}{=} \mathsf{Enabled}(i) \overset{v_i \in G_0}{<} \mathsf{Enabled}(i+1) \overset{d_{i+1}=1}{=} \mathsf{Executed}(i+1) \overset{(v_i, v_{i+1}) \notin E_0}{=} \mathsf{Executed}(i)$$

(2) $d_i = 1$, $d_{i+1} = 0$, $(v_i, v_{i+1}) \notin E_0$.

$$\mathsf{Executed}(i) \overset{d_i=1}{=} \mathsf{Enabled}(i) \overset{\text{taut.}}{\leq} \mathsf{Enabled}(i+1) \overset{d_{i+1}=0}{<} \mathsf{Executed}(i+1) \overset{(v_i, v_{i+1}) \notin E_0}{=} \mathsf{Executed}(i)$$

For the redefined version of $\tilde{M}$, in two cases $\tilde{M}((v_i, d_i), (v_{i+1}, d_{i+1})) = L_0(v_i, v_{i+1}) * M(v_i, v_{i+1})$ instead of $\tilde{M}((v_i, d_i), (v_{i+1}, d_{i+1})) = M(v_i, v_{i+1})$. In both cases, $(v_i, v_{i+1}) \in E_0$ (and thus $v_i \in G_0$) and $d_i = 1$. Therefore, $\mathsf{Executed}(i) = \mathsf{Enabled}(i) = i$. By definition of $\mathsf{Executed}$, it follows that $s_i \leq L_0(v_i, v_{i+1}) * M(v_i, v_{i+1})$ as required.

[A3] **implies** [B3]

From [B2], $0 < s_i \leq \tilde{M}((v_i, d_i), (v_{i+1}, d_{i+1}))$. Thus, by definition of $\mathsf{F}(\tilde{E}, \tilde{L}) \in \tilde{\mathcal{F}}$, it follows immediately that

(1) $((v_i, d_i), (v_{i+1}, d_{i+1})) \in \tilde{E}$ **iff** $(v_i, v_{i+1}) \in E$ for all $\mathsf{F}(E, L) \in \mathcal{F} - \{\mathsf{SF}(E_0, L_0)\}$ and

(2) $(v_i, d_i) \in \tilde{G}$ implies $v_i \in G$.

[5]Without this additional weak fairness condition one might simply choose $d_i = 0$ for all i.

The second condition is necessary for the fairness conditions only: if infinitely often (continuously) $(v_i, d_i) \in \tilde{G}$, then infinitely often (continuously) $v_i \in G$ and therefore infinitely often $(v_i, v_{i+1}) \in E$ from which due to (1) infinitely often $((v_i, d_i), (v_{i+1}, d_{i+1})) \in \tilde{E}$ can be deduced. Now two cases are to be distinguished:
For the original definition of $\tilde{M}$ nothing else has to be shown , because furthermore $\tilde{L}((v_i, d_i), (v_{i+1}, d_{i+1})) = L(v_i, v_{i+1})$ for all $(v_i, v_{i+1}) \in E$.
For the redefined version of $\tilde{M}$, it is possible that $\tilde{M} < M$, but only if $(v_i, v_{i+1}) \in E_0$. From the additional precondition, it then follows that besides $(v_i, v_{i+1}) \in E$ also $s_i \leq \tilde{M}((v_i, 1), (v_{i+1}, 0)) = M(v_i, v_{i+1}) * L_0(v_i, v_{i+1}) \leq M(v_i, v_{i+1}) * L(v_i, v_{i+1}) \leq L(v_i, v_{i+1})$.

It remains to show that the two extra fairness conditions are fulfilled. Suppose WF_{mirror} does not hold, that is, eventually always $d_i = 0$. From the definition of d_i, it follows that $\mathsf{Enabled}(i) < \mathsf{Executed}(i)$ for all $i > i_0$ for some i_0. Especially $\mathsf{Enabled}(i) \neq \omega$ and thus infinitely often $v_i \in G_0$. On the other hand, $(v_i, v_{i+1}) \notin E_0$ for all $i > i_0$ (otherwise $\mathsf{Enabled}(i_1) = \mathsf{Executed}(i_1)$ for some $i_1 > i_0$). This contradicts $\mathsf{SF}(E_0, L_0)$. WF_{mimic} follows immediately from $\mathsf{SF}(E_0, L_0)$.

Traces$(\tilde{\Sigma}) \subseteq$ **Traces**(Σ) :

For the opposite direction, given a sequence $((v_0, d_0), (v_1, d_1), \dots) \in (V \times \{0, 1\})^\omega$, it is straightforward to choose $(v_0, v_1, \dots) \in V^\omega$ as a corresponding run in Σ.

[B1] **implies** [A1] trivial

[B2] **implies** [A2]

By definition $\tilde{M}((v_1, d_1), (v_2, d_2)) \leq M(v_1, v_2)$. Thus $s_i \leq M(v_i, v_{i+1})$.

[B3] **implies** [A3]

For Buechi conditions $\mathsf{Büchi}(E, L) \in \mathcal{F}$, one has to prove that always eventually $(v_i, v_{i+1}) \in E$ and $s_i \leq L(v_i, v_{i+1})$. But because at least $((v_i, 0), (v_{i+1}, 0)) \in \tilde{E}$ it is guaranteed by $\mathsf{Büchi}(\tilde{E}, \tilde{L}) \in \tilde{\mathcal{F}}$ that always eventually $((v_i, 0), (v_{i+1}, 0)) \in \tilde{E}$ and $s_i \leq \tilde{L}((v_i, 0), (v_{i+1}, 0)) = L(v_i, v_{i+1})$.

The fairness conditions can only be proven by means of the additional preconditions. Then, only two restricted situations are to be dealt with:

Case 1: $G \cap G_0 = \emptyset$

For this case, it is sufficient to show that $v_i \in G$ implies $(v_i, d_i) \in \tilde{G}$. Then, it follows that if infinitely often (continuously) $v_i \in G$, then infinitely often (continuously) $(v_i, d_i) \in \tilde{G}$ and therefore infinitely often $((v_i, d_i), (v_{i+1}, d_{i+1})) \in \tilde{E}$ and $s_i \leq \tilde{L}((v_i, d_i), (v_{i+1}, d_{i+1}))$ such that by definition of $\tilde{E}$ infinitely often $(v_i, v_{i+1}) \in E$ and $s_i \leq L(v_i, v_{i+1})$ can be deduced.

Thus, suppose $v_i \in G$. Then, $v_i \notin G_0$ and by definition of $\tilde{M}$ it follows that $0 < s_i \leq \tilde{M}((v_i, d_i), (v_{i+1}, d_{i+1})) = M(v_i, v_{i+1})$ and $d_{i+1} = d_i$. On the other hand, $v_i \in G$ implies the existence of a state v such that $(v_i, v) \in E$. Therefore, from the definition of $\tilde{E}$ it follows that $((v_i, d_i), (v, d_i)) \in \tilde{E}$ and thus $(v_i, d_i) \in \tilde{G}$ as required.

Case 2: $v_i \in G \cap G_0$ and $(v_i, v_{i+1}) \in E_0$ implies $(v_i, v_{i+1}) \in E$ and $L_0(v_i, v_{i+1}) \leq L(v_i, v)$

In this case, all reasoning takes place in the original BTS: If infinitely often (continuously) $v_i \in G$, then infinitely often $v_i \in G_0$. Then, due to $\mathsf{SF}(E_0, L_0)$ (which is proven below), infinitely often $(v_i, v_{i+1}) \in E_0$ and $s_i \leq L_0(v_i, v_{i+1})$ and thus infinitely often $(v_i, v_{i+1}) \in E$ and

$s_i \leq L(v_i, v_{i+1})$.

It remains to show $\mathsf{SF}(E_0, L_0)$. Suppose not $\mathsf{SF}(E_0, L_0)$. That is, infinitely often $v_i \in G_0$ but from some point i_0 on $(v_i, v_{i+1}) \notin E_0$ or $s_i * L_0(v_i, v_{i+1}) = 0$. Two cases may arise:

Case 1: there is at least one $i \geq i_0$ such that $v_i \in G_0$ and $d_i = 1$

Then from the redefined version of $\tilde{M}$, it follows that either

Case 1.1: $v_{i+1} = v_i$ and $d_{i+1} = 1$ and $(v_i, v_{i+1}) \notin E_0$

or

Case 1.2: $(v_i, v_{i+1}) \in E_0$ and $s_i \leq \tilde{M}((v_i, 1), (v_{i+1}, 0)) = M(v_i, v_{i+1}) * L_0(v_i, v_{i+1})$

Case 1.2 is an immediate contradiction. Thus continuously case 1.1. But that contradicts WF_{mimic}.

Case 2: for all $i \geq i_0$ such that $v_i \in G_0$ it holds that $d_i = 0$

Due to WF_{mirror}, there is some $j > i$ such that $d_j = 1$. But from the definition of $\tilde{M}$, it follows that the only possibility to reach the next $k > j$ with $v_k \in G_0$ and $d_k = 0$ is by following an edge with $(v, v') \in E_0$.

■

Without the additional preconditions, trace equality can not be obtained. In the general situation, difficulties may arise due to "conflicting" fairness conditions $\mathsf{F}(G, E, L)$ as sketched in Figure 4.5.

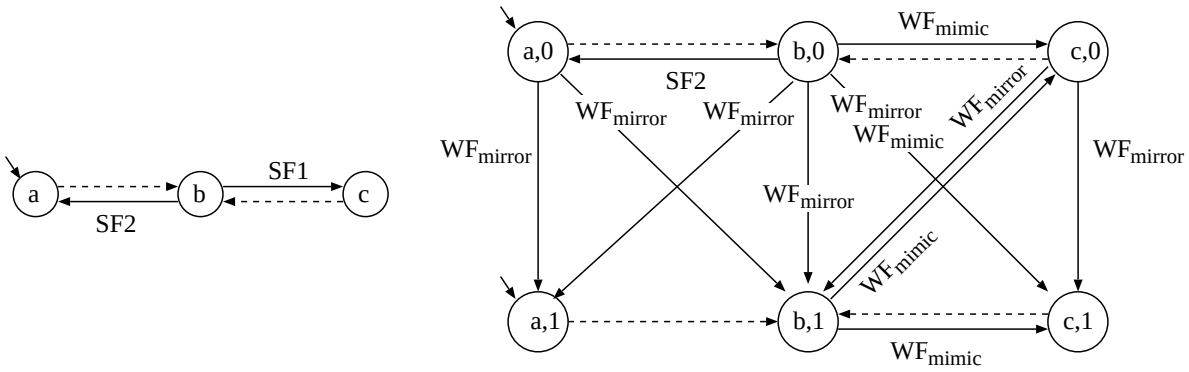


Figure 4.5: The strong fairness condition $\mathsf{SF1}$ of the left BTS is replaced but not "implemented" on the right. For example, any trace running through the states $((\mathsf{a}, 0), (\mathsf{b}, 1), (\mathsf{c}, 0), (\mathsf{b}, 1), (\mathsf{c}, 0), ...)$ obeys the strong fairness condition $\mathsf{SF2}$ in the BTS on the right. However, the projection $(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{b}, \mathsf{c}, ...)$ of this run is not strong fair with respect to $\mathsf{SF2}$ in the left BTS.

Applying this construction successively to all $\mathsf{SF}(E, L) \in \mathcal{F}$ results in a BTS without strong fairness conditions and such that all properties that hold for the resulting BTS also hold in the original BTS. If furthermore the additional condition holds then the resulting BTS even is trace-equivalent to the original one.

4.2.4 Conjunction

From Corollary 4.2.6, it follows that weak fairness conditions can be expressed as Büchi conditions for arbitrary traces. In the previous section, it was shown that strong fairness can be replaced by weak fairness. Now, in order to define the conjunction of two BTS, it is assumed that all acceptance conditions of a BTS are Büchi conditions.

Definition 4.2.8 (Conjunction)
Let Σ_1 and Σ_2 be two BTS $\Sigma_i = (V_i, I_i, \mathcal{B}, st_i, M_i, \mathcal{F}_i)$ *such that*

(1) $\mathcal{F}_i$ contain only Büchi conditions, and
(2) [St_Independence] $0 < st_1 * st_2 * M_1(e_1) * M_2(e_2)$
for any edges with $0 < st_1 * M_1(e_1)$ *and* $0 < st_2 * M_2(e_2)$.

Then their conjunction (product) is defined as

$$\Sigma_1 \wedge \Sigma_2 \;=\; (V_1 \times V_2, I_1 \times I_2, \mathcal{B}, st, M, \mathcal{F})$$

where

$$st \stackrel{def}{=} st_1 * st_2$$

and

$$M((v_1, v_2), (v_1{}', v_2{}')) \stackrel{def}{=} M_1(v_1, v_1{}') * M_2(v_2, v_2{}')$$

and

$$\mathcal{F} \stackrel{def}{=} \{\mathsf{Büchi}(\tilde{E_1}, \tilde{L_1}) \mid \mathsf{Büchi}(E_1, L_1) \in \mathcal{F}_1\} \cup \{\mathsf{Büchi}(\tilde{E_2}, \tilde{L_2}) \mid \mathsf{Büchi}(E_2, L_2) \in \mathcal{F}_2\}$$

with $\tilde{E_1} \stackrel{def}{=} \{((v_1, v_2), (v_1{}', v_2{}')) \in (V_1 \times V_2)^2 \mid (v_1, v_1{}') \in E_1$ *and*
$L_1(v_1, v_1{}') * M((v_1, v_2), (v_1{}', v_2{}')) \neq 0\ \}$

$\tilde{L_1}((v_1, v_2), (v_1{}', v_2{}')) \stackrel{def}{=} L_1(v_1, v_1{}') * M((v_1, v_2), (v_1{}', v_2{}'))$

and $\tilde{E_2} \stackrel{def}{=} \{((v_1, v_2), (v_1{}', v_2{}')) \in (V_1 \times V_2)^2 \mid (v_2, v_2{}') \in E_2$ *and*
$L_2(v_2, v_2{}') * M((v_1, v_2), (v_1{}', v_2{}')) \neq 0\}$

$\tilde{L_2}((v_1, v_2), (v_1{}', v_2{}')) \stackrel{def}{=} M((v_1, v_2), (v_1{}', v_2{}')) * L_2(v_2, v_2{}')$

$\Sigma_1 \wedge \Sigma_2$ is well-defined: [Init], [St] and [Fair] follow immediately from the definition.

Corollary 4.2.9
$\wedge$ is commutative and associative

Thus one may write $\Sigma_1 \wedge \Sigma_2 \wedge \Sigma_3$ for both $(\Sigma_1 \wedge \Sigma_2) \wedge \Sigma_3$ and $\Sigma_1 \wedge (\Sigma_2 \wedge \Sigma_3)$. Further, this definition generalizes trivially to conjunctions of BTS indexed over a finite set, written as $\prod_{i \in I} \Sigma_i$. For this case, the condition requiring the independence of stutter steps may be summarized to

[St_Independence]
$0 < \prod_{i \in I} st_i * M_i(e_i)$ for any set of edges $e_i \in V_i \times V_i$ such that $0 < st * M_i(e_i)$

The traces of $\Sigma_1 \wedge \Sigma_2$ are given as the intersection of the traces of Σ_1 and the traces of Σ_2.

Theorem 4.2.10 (Conjunction Theorem)

$$\mathbf{Traces}(\Sigma_1 \wedge \Sigma_2) = \mathbf{Traces}(\Sigma_1) \cap \mathbf{Traces}(\Sigma_2)$$

Proof:

Let $(s_0, s_1, \dots) \in S(\mathcal{B})^\omega$.

Summarizing the definitions for runs and traces,

$(s_0, s_1, \dots) \in \mathbf{Traces}(\Sigma_1) \cap \mathbf{Traces}(\Sigma_2)$ **iff** there are sequences $(v_0, v_1, \dots) \in V_1^\omega$ and $(w_0, w_1, \dots) \in V_2^\omega$ such that

[A1] $v_0 \in I_1$ and $w_0 \in I_2$

[A2] $s_i \leq M_1(v_i, v_{i+1})$ and $s_i \leq M_2(w_i, w_{i+1})$

[A3] for all $\mathsf{Büchi}(E_1, L_1) \in \mathcal{F}_1$: Infinitely often $s_i \leq L_1(v_i, v_{i+1})$ and $(v_i, v_{i+1}) \in E_1$
(and analog for $\mathsf{Büchi}(E_2, L_2) \in \mathcal{F}_2$)

$(s_0, s_1, \dots) \in \mathbf{Traces}(\Sigma_1 \wedge \Sigma_2)$ **iff** there is a sequence $((v_0, w_0), (v_1, w_1), \dots) \in (V_1 \times V_2)^\omega$ such that

[B1] $(v_0, w_0) \in I_1 \times I_2$

[B2] $s_i \leq M((v_i, w_i), (v_{i+1}, w_{i+1})) = M_1(v_i, v_{i+1}) * M_2(w_i, w_{i+1})$

[B3] for all $\mathsf{Büchi}(\tilde{E}, \tilde{L}) \in \mathcal{F}$: Infinitely often $s_i \leq \tilde{L}((v_i, w_i), (v_{i+1}, w_{i+1}))$ and $((v_i, w_i), (v_{i+1}, w_{i+1})) \in \tilde{E}$.

Obviously, [A1] **iff** [B1] and [A2] **iff** [B2] (compare Lemma 4.1.5).

Instead of proving [A3] **iff** [B3] directly, the proof obligation is strengthened to

$$(v_i, v_{i+1}) \in E_1 \text{ and } s_i \leq L_1(v_i, v_{i+1})$$
$$\textbf{iff } ((v_i, w_i), (v_{i+1}, w_{i+1})) \in \tilde{E_1} \text{ and } s_i \leq \tilde{L_1}((v_i, w_i), (v_{i+1}, w_{i+1}))$$

To show the direction from left to right, first note that $s_i \leq M_1(v_i, v_{i+1}) * M_2(w_i, w_{i+1})$ by [A2]. Thus $s_i \leq L_1(v_i, v_{i+1}) * M((v_i, w_i), (v_{i+1}, w_{i+1})) = \tilde{L_1}((v_i, w_i), (v_{i+1}, w_{i+1}))$.

From the definition of E it then follows that $((v_i, w_i), (v_{i+1}, w_{i+1})) \in \tilde{E}$.

The direction from right to left follows immediately by the definitions of E and L.

■

As already mentioned, it is not possible to base conjunction on fairness conditions in a way similar straightforward. Figure 4.6 exemplifies that weak fairness conditions of components may eventually not be expressed by weak fairness conditions in the composed BTS.

4.2.5 Executable Boolean Transition Systems

This section presents a first result concerning executable BTS, namely that BTS without Büchi-acceptance conditions are executable.

Theorem 4.2.11 *Each BTS Σ without* Büchi*-acceptance conditions is executable. Moreover, all initial runs can be completed.*

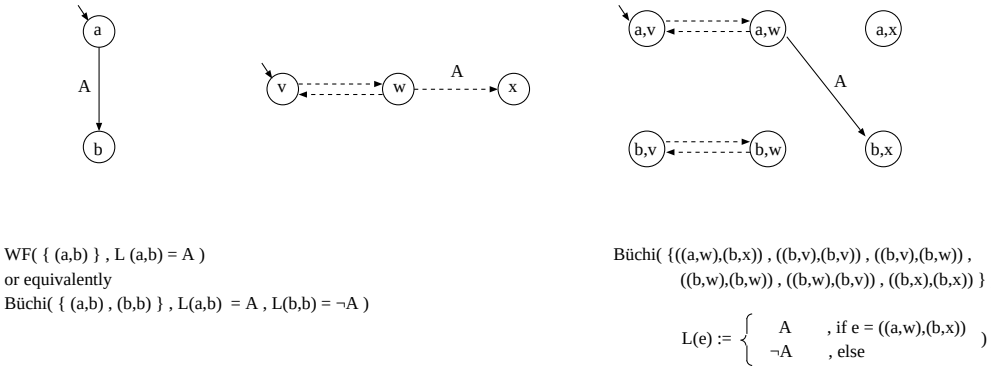


Figure 4.6: Σ_1 (on the left) does not allow the trace $(\neg A, \neg A, \neg A, \dots)$ that would be possible in Σ_2 (in the middle). To disallow this trace in the composition, one has to add a Büchi-condition that forces the transition from state (a, w) to state (b, x). This Büchi-condition can not be expressed by a weak fairness condition: $\mathsf{WF}(\ \{((a,w),(b,x))\}\ ,\ L((a,w),(b,x)) = A\ )$ does not disallow alternating continuously between (a, v) and (a, w). One may be tempted to require $\mathsf{SF}(\ \{((a,w),(b,x))\}\ ,\ L((a,w),(b,x)) = A\ )$ instead. This would work for this example but it be too strong (in the sense of excluding too many traces in the composition) in other examples (one such example may be obtained by replacing the fairness constraint $\mathsf{WF}(\ \{(a,b)\}\ ,\ L(a,b) = A\ )$ for Σ_1 by $\mathsf{WF}(\ \{(w,x)\}\ ,\ L(w,x) = A\ )$ for Σ_2. Then both Σ_1 and Σ_2 allow the trace $(\neg A, \neg A, \neg A, \dots)$ that would be forbidden by the strong fairness condition $\mathsf{SF}(\ \{((a,w),(b,x))\}\ ,\ L((a,w),(b,x)) = A\ )$ in the composition).

Proof:

The idea of the proof is simple: for each weak fairness conditions $\mathsf{WF}(E, L)$ one counts how long it has been enabled continuously in the past without taking an edge from E. For each strong fairness conditions $\mathsf{SF}(E, L)$ one counts how often it was enabled since the last time an edge from E was taken. Then, a fairness condition that is enabled and "waits" for the longest time is chosen and an edge $e \in E$ is executed. Since there are only finitely many fairness condition, this leads to a run fulfilling all fairness conditions.

More formally, let $\mathcal{F} = \{\mathsf{F}_1(G_1, E_1, L_1), \dots \mathsf{F}_N(G_N, E_N, L_N)\}$ with $\mathsf{F_k} \in \{\mathsf{WF}, \mathsf{SF}\}$ for all $1 \le k \le N$. Then a run $(v_0, v_1, \dots)$ may be determined inductively as follows:

v_0 may be chosen arbitrarily from I.

Now, suppose $v_0, v_1, \dots, v_n$ are already determined and define a function $Wait : |\mathcal{F}| \times \mathbb{N} \to \mathbb{N}$ by

$$Wait(k,i) \stackrel{\text{def}}{=} \begin{cases} \max\{j \mid v_{i-j}, \dots v_i \in G_k \text{ and either } j = 0 \text{ or} \\ \qquad j \ge 1 \text{ and} (v_{i-j}, v_{i-(j-1)}), \dots, (v_{i-1}, v_i) \notin E_k\} & , \quad v_i \in G_k \text{ and } \mathsf{F_k} = \mathsf{WF} \\ \#\{j \mid last(E_k, i) \le j < i \text{ and } v_j \in G_k\ \} & , \quad v_i \in G_k \text{ and } \mathsf{F_k} = SF \\ -1 & , \quad v_i \notin G_k \end{cases}$$

where $last(E,i) \stackrel{\text{def}}{=} \begin{cases} \max\{j \mid (v_{j-1}, v_j) \in E \text{ and } j \le i\ \} & , \text{ exists } j \le i \text{ with } (v_{j-1}, v_j) \in E \\ 0 & , \text{ else} \end{cases}$.

Then two cases can be distinguished:

1. $v_n \notin \bigcup_k G_k$

   Then any $v_{n+1} \in V$ may be chosen such that $M(v_n, v_{n+1}) \neq 0$, e.g. $v_{n+1} \stackrel{\text{def}}{=} v_n$ ([St] then guarantees that $M(v_n, v_n) \neq 0$).

2. $v_n \in \bigcup_k G_k$

   Choose a k_0 such that $Wait(k_0, n)$ is maximal among all $Wait(k, n)$ (then $v_n \in G_{k_0}$ by definition). Due to the definition of the G_k, there exists a v_{n+1} such that $(v_n, v_{n+1}) \in E_{k_0}$ and, due to [Fair], $M(v_n, v_{n+1}) \neq 0$ as required in [Run2].

Given an initial run $(v_0, v_1, \ldots, v_n)$, the same decision procedure is used to determine $v_{n+1}, v_{n+2}, \ldots$. ■

4.2.6 Executable Composition, Delays

In this section the composition of BTS naming only weak fairness conditions is further investigated. Summarizing the previous sections, it was shown that

1. For BTS containing only Büchi conditions, the composition theorem holds:
   $\mathbf{Traces}(\Sigma_1 \wedge \Sigma_2) = \mathbf{Traces}(\Sigma_1) \cap \mathbf{Traces}(\Sigma_1)$
2. BTS containing only fairness conditions are executable: $\mathbf{Traces}(\Sigma) \neq \emptyset$
3. Strong fairness conditions can be replaced and often even implemented by weak fairness conditions. To simplify the exposition in this section, it is assumed that there are no strong fairness conditions.

As weak fairness conditions can trivially be expressed as Büchi conditions,
$\mathbf{Traces}(\Sigma_1 \wedge \Sigma_2) = \mathbf{Traces}(\Sigma_1) \cap \mathbf{Traces}(\Sigma_1)$
holds especially for BTS Σ_1 and Σ_2 containing only weak fairness conditions. However, even if both Σ_1 and Σ_2 are executable, it is still possible that their composition is not, that is, $\mathbf{Traces}(\Sigma_1 \wedge \Sigma_2) = \emptyset$ as exemplified in Figure 4.7.

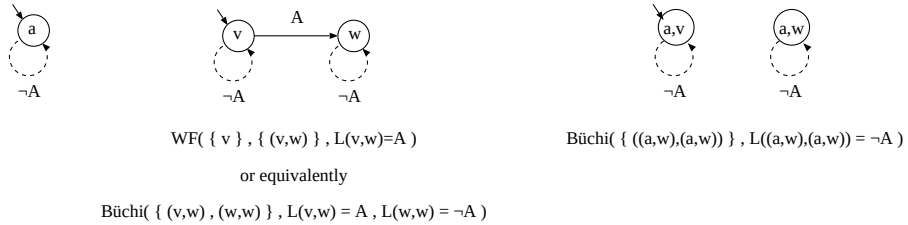


Figure 4.7: The BTS shown on the left has $(\neg A, \neg A, \ldots)$ as sole trace. The set of traces of the BTS shown in the middle is described by $(\underbrace{\neg A, \neg A, \ldots, \neg A}_{\text{n times}}, A, \neg A, \neg A, \ldots)$. That is, the fairness condition forces a step labeled by A. Thus the intersection of both sets of traces is empty. In the BTS for the composition (shown on the right) the Büchi condition can not be fulfilled as the state (a, w) is unreachable. Thus $\mathbf{Traces}(\Sigma_1) \cap \mathbf{Traces}(\Sigma_1) = \mathbf{Traces}(\Sigma_1 \wedge \Sigma_2) = \emptyset$.

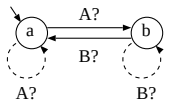
In the example, a fairness condition requires the occurrence of an action A in each trace of one BTS whereas the other BTS never allows the occurrence of this action. To overcome

such situation, a notion of being input enabled was introduced in the context of I/O-automata. Transcribed to TLT, this means that each BTS in all states must "accept" occurrences of actions that other modules require to be fair. [6] Formally,

Definition 4.2.12 (Input Enabledness)
Let $\Sigma = (V, I, \mathcal{B}, st, M, \mathcal{F})$, $\tilde{\Sigma} = (W, J, \mathcal{B}, st, N, \mathcal{G})$, *and* $\mathsf{WF}(G, E, L) \in \mathcal{G}$.

Then the BTS Σ *is called input enabled wrt the (weak) fairness condition* $\mathsf{WF}(G, E, L)$ **iff**
for all $w \in G$ *and all* $v \in V$:
there exist some $v' \in V$ *and* $w' \in W$ *such that* $(w, w') \in E$ *and* $0 < L(w, w') * M(v, v')$

However, this poses strong restrictions on the specifications. One drawback is that modules can not agree on a protocol restricting the states where edges $e \in E$ may occur. Such situations will be dealt in Chapter 5 in the context of assumption-commitment reasoning. A second drawback is, that the implementation is hindered: consider a BTS that is supposed to accept two independent actions A and B. On a low abstraction level it may only be possible to deal with one "input" action at a time. For example, the BTS may alternate between two states as shown in Figure 4.8. Although not being input enabled, this BTS guarantees that each of the actions A and B can always eventually occur. Expressed otherwise, the acceptance of the actions is at most delayed, because the BTS eventually will reach a state where it can accept them. A subtle point is that in order to guarantee to reach these states, the BTS itself uses (weak) fairness conditions. However, these fairness conditions are not "seen" from the outside and therefore do not restrict the environment[7]: the environment "accepts" these fairness conditions for example by simply stuttering (which is possible in all states).



WF({ (a,b) , (b,a) } , L(a,b) = ¬B , L(b,a)=¬A)

Figure 4.8: The shown BTS is not input enabled: In state a it does not accept the action B, in state b it does not accept A.

Definition 4.2.13 (Delays)
Let $\Sigma = (V, I, \mathcal{B}, st, M, \mathcal{F})$, $\tilde{\Sigma} = (W, J, \mathcal{B}, st, N, \mathcal{G})$, *and* $\mathsf{WF}(G, E, L) \in \mathcal{G}$.

Then the BTS Σ *at most delays the (weak) fairness condition* $\mathsf{WF}(G, E, L)$ **iff**

- *[Delay1] for all* $w \in G$ *and all* $v \in V$:
  there exist some $v' \in V$ *and* $w' \in W$ *such that* $(w, w') \in E$ *and* $0 < L(w, w') * M(v, v')$
  or
  $0 < M(v, v')$ *implies* $0 < st * M(v, v')$ *for arbitrary* $v' \in V$.
- *[Delay2] for all* $w \in G$ *and all* $(v_0, v_1, \dots) \in \mathbf{Runs}(\Sigma)$:
  infinitely often there exist some $v' \in V$ *and* $w' \in W$ *such that* $(w, w') \in E$ *and* $0 < L(w, w') * M(v_i, v')$

[6] There is a subtle difference however in that in the theory of I/O-automata, all inputs have to be accepted in all states regardless whether the environment "insists" (by means of a fairness condition) to do them.
[7] Otherwise, there would be cyclic fairness conditions that might result in live-locks.

Clearly, the notion of "at most delaying" subsumes "being input enabled":

Corollary 4.2.14
If a BTS $\Sigma = (V, I, \mathcal{B}, st, M, \mathcal{F})$ is input enabled with respect to a (weak) fairness condition $\mathsf{WF}(G, E, L)$ then it at most delays it.

Intuitively, [Delay1] requires that a BTS in any state either
(1) is input enabled for a given fairness condition, or
(2) allows a stutter step on any edge leaving this state .
Note, however, that [Delay1] only requires the existence of a stutter step in this second case and does not force one. On the contrary, the BTS might even take part in a synchronization.

[Delay2] simply states that the BTS infinitely often has to be input enabled wrt the fairness condition. Typically, this is guaranteed by means of so-called local (weak) fairness conditions:

Definition 4.2.15 (Local Fairness)
Let $\Sigma = (V, I, \mathcal{B}, st, M, \mathcal{F})$ and $\mathsf{WF}(G, E, L) \in \mathcal{F}$.

If $st \leq L(v, v')$ for all $(v, v') \in E$ then $\mathsf{WF}(G, E, L)$ is called local weak fairness condition.

If $\hat{\Sigma}$ is another BTS, then from [St] it follows, that $\hat{\Sigma}$ may accept any local fairness condition simply by doing a stutter step.

Corollary 4.2.16
Let $\Sigma = (V, I, \mathcal{B}, st, M, \mathcal{F})$, $\hat{\Sigma} = (W, J, \mathcal{B}, st, N, \mathcal{G})$, and $\mathsf{WF}(G, E, L) \in \mathcal{F}$.

If $\mathsf{WF}(G, E, L)$ is a local weak fairness condition then $\hat{\Sigma}$ is input enabled wrt (and thus at most delays) $\mathsf{WF}(G, E, L)$.

Theorem 4.2.17 (Executable Conjunction)
Let $\Sigma^1 = (V^1, I^1, \mathcal{B}, st, M^1, \mathcal{F}^1), \ldots, \Sigma^N = (V^N, I^N, \mathcal{B}, st, M^N, \mathcal{F}^N)$ be BTS such that
(1) all acceptance conditions are weak fairness conditions,
(2) the BTS at most delay each others fairness conditions,
(3) [St_Independence]

$$0 < \prod_{1 \leq n \leq N} st * M^n(e^n) \qquad \text{for any set of edges } e^n \in V^n \times V^n$$
$$\text{such that } 0 < st^n * M^n(e^n) \text{ for all } 1 \leq n \leq N$$

(4) [Fair_Independence] Let $v^n \in V^n$ and in addition for some fixed but arbitrary k let $v^k \in G$ for some $\mathsf{WF}(G, E, L) \in \mathcal{F}^k$.
If for all $n \neq k$ there are $w^n \in V_n$ and $w^k \in V_k$ such that $0 < L(v^k, w^k) * M^n(v^n, w^n)$ then it holds that

$$0 < L(v^k, \tilde{w}^k) * \prod_{1 \leq n \leq N} M^n(v^n, \tilde{w}^n) \text{ for some } \tilde{w}^n \in V_n.$$

Then $\mathbf{Traces}(\Sigma^1 \wedge \cdots \wedge \Sigma^N) = \mathbf{Traces}(\Sigma^1) \cap \cdots \cap \mathbf{Traces}(\Sigma^N) \neq \emptyset$

Proof(sketched):

1. Determine a fairness condition $\mathsf{WF}(G, E, L) \in \mathcal{F}^k$ that is enabled the longest without being executed. Further determine an edge $e^k \in E$ that Σ^k "wants to take", that is, such that $0 < L(e^k) * M^k(e^k)$.

2. Continue the runs of all BTS simultaneously:
   If $L(e^k)$-labeled steps are possible in all Σ^n, that is, if there are edges e^n in these components such that $0 < L(e^k) * M^n(e^n)$, then due to [Fair_Independence] one may choose some $s \leq L(\tilde{e}^k) * \prod_{1 \leq n \leq N} M^n(\tilde{e}^n)$. Go back to 1.
   If not, choose arbitrary edges $e^n = (v^n, w^n)$ along some run in that component such that $0 < st * M(e^n)$ in all components where $0 < L(e^k) * M^n(e^n)$ does not (yet) hold for any e^n (the existence of these edges is guaranteed by [Delay1]). The remaining components (including Σ^k) do not change their local state v^n (they choose the edge $e^n = (v^n, v^n)$). For these components $0 < st * M(e^n)$ holds too (due to condition [St] in the definition of BTS). Because of [St_Independence], it is then possible to choose some $s \leq \prod_{1 \leq n \leq N} st^n * M^n(e^n)$.

Termination of step 2 follows from [Delay2]: all Σ^n reach after a finite sequence of steps a state v^n where some $L^k(e^k)$-labeled step is possible. Then they stay at state v^n until the last component reaches a state where some $L^k(e^k)$-labeled step is possible.

■

REMARK 1. All definitions in this section can be extended easily to also include strong fairness conditions. Then, $\mathbf{Traces}(\Sigma^1) \cap \cdots \cap \mathbf{Traces}(\Sigma^N) \neq \emptyset$ can still be proven by generalizing the first sentence in the proof of Theorem 4.2.17 to:
"Determine a fairness condition $\mathsf{F}(G, E, L) \in \mathcal{F}^k$ that waits for the longest time"
Here, "waiting" refers to the function $Wait(k, i)$ defined in the proof of Theorem 4.2.5.
However, the definition of the conjunction of BTS may no longer be traced back to Definition 4.2.8. Nevertheless, it is of course possible to define $\mathbf{Traces}(\Sigma^1 \wedge \cdots \wedge \Sigma^N)$ semantically as the intersection of the traces of the BTS. Then, the whole conclusion of the theorem, namely
$\mathbf{Traces}(\Sigma^1 \wedge \cdots \wedge \Sigma^N) = \mathbf{Traces}(\Sigma^1) \cap \cdots \cap \mathbf{Traces}(\Sigma^N) \neq \emptyset$
holds for arbitrary fairness conditions.

4.2.7 Systems

By building systems, typically the labeling Boolean algebras of the components differ from each other. For example, some components may run completely independent. Thus, the Boolean algebras of the components have to be "embedded" into the larger Boolean algebra of the composed system.

To compute the composition of two BTS's on different Boolean algebras $\mathcal{B}_1$ and $\mathcal{B}_2$, it is necessary to obtain a Boolean algebra $\mathcal{B}$ and two algebra monomorphisms $\varphi_i : \mathcal{B}_i \hookrightarrow \mathcal{B}$. That is, the BTS's are projected to $\mathcal{B}$ before they are conjoined. The resulting product thus depends on the chosen φ_i. Formally, the image of a BTS under a Boolean algebra monomorphism is given by the following construction:

Definition 4.2.18 (Image)

Let $\Sigma_1 = (V, I, \mathcal{B}_1, st_1, M_1, \mathcal{F}_1)$ be a BTS *over $\mathcal{B}_1$ and φ an algebra monomorphism $\varphi : \mathcal{B}_1 \to \mathcal{B}$.*

Then the image of Σ_1 under φ is the BTS *$\varphi\Sigma_1 \stackrel{def}{=} (V, I, \mathcal{B}, st, M, \mathcal{F})$ with*
$M(v, v') \stackrel{def}{=} \varphi(M_1(v, v'))$, $st \stackrel{def}{=} \varphi(st_1)$ and $\mathcal{F} \stackrel{def}{=} \{\mathsf{F}(E, \tilde{L}) \mid \mathsf{F}(E, L) \in \mathcal{F}_1$ and $\tilde{L}(e) = \varphi(L(e))\ \}$.

That the image of a BTS Σ_1 is well-defined follows from the simple fact that monomorphisms map non-zero elements to non-zero elements (Lemma 4.1.14). Thus [St] and [Fair] also hold in $\varphi\Sigma_1$.

The following lemma states that the set of runs is not at all influenced by this construction:

Lemma 4.2.19 $\quad$ $\mathbf{Runs}(\varphi\Sigma) = \mathbf{Runs}(\Sigma)$

Proof:

Only the second condition [Run2] in definition 4.2.3 is influenced by φ. But since φ is injective, $M(v_i, v_{i+1}) \neq 0$ **iff** $\varphi(M(v_i, v_{i+1})) \neq 0$ holds trivially.

■

However, the number of traces in general will increase due to the fact that the number of atoms increases: $|S(\mathcal{B})| \leq |S(\varphi(\mathcal{B}))|$. Still, it is possible to relate the traces in case that $\mathcal{B} = \mathcal{P}(X \times Y)$:

Lemma 4.2.20
Let $\Sigma_X = (V, I, \mathcal{P}(X), st, M, \mathcal{F})$ *and* $\varphi_X \ : \ \mathcal{P}(X) \hookrightarrow \mathcal{P}(X \times Y)$. *Then*

1. $\mathbf{Traces}(\Sigma_X) \ = \ \mathbf{Traces}(\varphi_X\Sigma_X)\big|_X$ *and*
2. $\sigma \in \mathbf{Traces}(\varphi_X\Sigma_X)$ **iff** $\sigma\big|_X \in \mathbf{Traces}(\Sigma_X)$.

Proof:

For the proof it is assumed that all acceptance conditions are Büchi-conditions. The proof for fairness conditions is similar.

$(x_0, x_1, \dots) \in \mathbf{Traces}(\Sigma_X)$
iff exists $(v_0, v_1, \dots) \in \mathbf{Runs}(\Sigma_X)$ such that
[1] $\{x_i\} \subseteq M_X(v_i, v_{i+1})$ for all i
[2] infinitely often $\{x_i\} \subseteq L_X(v_i, v_{i+1})$ and $(v_i, v_{i+1}) \in E_X$ for all $\mathsf{Büchi}(E_X, L_X) \in \mathcal{F}_X$
iff exists $(v_0, v_1, \dots) \in \mathbf{Runs}(\Sigma_X)$ and exists $(y_0, y_1, \dots) \in Y^\omega$ such that
[1] $\{(x_i, y_i)\} \subseteq M_X(v_i, v_{i+1}) \times Y$ for all i
[2] infinitely often $\{(x_i, y_i)\} \subseteq L_X(v_i, v_{i+1}) \times Y$ and $(v_i, v_{i+1}) \in E_X$ for all $\mathsf{Büchi}(E_X, L_X) \in \mathcal{F}_X$
iff exists $(v_0, v_1, \dots) \in \mathbf{Runs}(\varphi_X\Sigma_X)$ and exists $(y_0, y_1, \dots) \in Y^\omega$ such that
[1] $\{(x_i, y_i)\} \subseteq M_X(v_i, v_{i+1}) \times Y$ for all i
[2] infinitely often $\{(x_i, y_i)\} \subseteq L_X(v_i, v_{i+1}) \times Y$ and $(v_i, v_{i+1}) \in E_X$ for all $\mathsf{Büchi}(E_X, L_X) \in \mathcal{F}_X$
iff exists $(v_0, v_1, \dots) \in \mathbf{Runs}(\varphi_X\Sigma_X)$ and exists $(y_0, y_1, \dots) \in Y^\omega$ such that
[1] $\{(x_i, y_i)\} \subseteq \varphi_X M_X(v_i, v_{i+1})$ for all i
[2] infinitely often $\{(x_i, y_i)\} \subseteq \varphi_X L_X(v_i, v_{i+1})$ and $(v_i, v_{i+1}) \in E_X$ for all $\mathsf{Büchi}(E_X, L_X) \in \mathcal{F}_X$
iff exists $(y_0, y_1, \dots) \in Y^\omega$ such that
$(\ (x_0, y_0), (x_1, y_1), \dots\ ) \in \mathbf{Traces}(\varphi_X\Sigma_X)$
iff $(x_0, x_1, \dots) \in \mathbf{Traces}(\varphi_X\Sigma_X)\big|_X$

Similarly, for the second proposition,

$((x_0, y_0), (x_1, y_1), \dots) \in \mathbf{Traces}(\varphi_X\Sigma_X)$
iff exists $(v_0, v_1, \dots) \in \mathbf{Runs}(\varphi_X\Sigma_X)$ such that
[1] $\{(x_i, y_i)\} \subseteq \varphi_X M_X(v_i, v_{i+1})$ for all i

[2] infinitely often $\{(x_i, y_i)\} \subseteq \varphi_X L_X(v_i, v_{i+1})$ and $(v_i, v_{i+1}) \in E_X$ for all $\mathsf{Büchi}(E_X, L_X) \in \mathcal{F}_X$

iff exists $(v_0, v_1, \dots) \in \mathbf{Runs}(\Sigma_X)$ such that

[1] $\{(x_i, y_i)\} \subseteq \varphi_X M_X(v_i, v_{i+1})$ for all i

[2] infinitely often $\{(x_i, y_i)\} \subseteq \varphi_X L_X(v_i, v_{i+1})$ and $(v_i, v_{i+1}) \in E_X$ for all $\mathsf{Büchi}(E_X, L_X) \in \mathcal{F}_X$

iff exists $(v_0, v_1, \dots) \in \mathbf{Runs}(\Sigma_X)$ and exists $(y_0, y_1, \dots) \in Y^\omega$ such that

[1] $\{x_i\} \subseteq M_X(v_i, v_{i+1})$ for all i

[2] infinitely often $\{x_i\} \subseteq L_X(v_i, v_{i+1})$ and $(v_i, v_{i+1}) \in E_X$ for all $\mathsf{Büchi}(E_X, L_X) \in \mathcal{F}_X$

iff $(x_0, x_1, \dots) \in \mathbf{Traces}(\Sigma_X)$

■

Especially, the latter lemma can be applied to Boolean transition systems that are based on algebras of transitions as introduced in Section 4.1.6. For example, if

$$X \stackrel{\text{def}}{=} \prod_{x \in \mathsf{Vars}} \mathcal{D}(x) \times \prod_{A \in \mathsf{Acts}} \mathcal{D}_\perp(A) \times \prod_{x \in \mathsf{Vars}} \mathcal{D}(x)$$

then $\mathcal{P}(X) = \mathcal{B}_{(\mathsf{Vars},\mathsf{Acts})}$ which can be embedded into any algebra $\mathcal{B}_{(\mathsf{Vars}_1,\mathsf{Acts}_1)}$ such that $\mathsf{Vars} \subseteq \mathsf{Vars}_1$ and $\mathsf{Acts} \subseteq \mathsf{Acts}_1$.

Definition 4.2.21 (Composition)
Let I be a finite index set and let $\Sigma_i = (V_i, I_i, \mathcal{B}_i, st_i, M_i, \mathcal{F}_i)$ be BTS without strong fairness conditions for all $i \in I$. Further assume some Boolean algebra $\mathcal{B}$ and monomorphisms $\varphi_i : \mathcal{B}_i \hookrightarrow \mathcal{B}$ such that [St_Independence] *holds for $\varphi_i \Sigma_i$.*

Then the composition $\|_{i \in I} \Sigma_i$ is defined by $\|_{i \in I} \Sigma_i \stackrel{def}{=} \bigwedge_{i \in I} \varphi_i \Sigma_i$.

Corollary 4.2.22

1. *If $\mathcal{B}_1 = \mathcal{B}_2 = \mathcal{B}$ then $\Sigma_1 \| \Sigma_2 = \Sigma_1 \wedge \Sigma_2$*
2. $\mathbf{Traces}(\|_{i \in I} \Sigma_i) = \bigcap_{i \in I} \mathbf{Traces}(\varphi_i \Sigma_i)$
3. *If all acceptance conditions are fairness conditions and the BTS $\varphi_i \Sigma_i$ at most delay each others fairness conditions and fulfill [Fair_Independence], then $\mathbf{Traces}(\|_{i \in I} \Sigma_i) \neq \emptyset$*

Example: Let Σ_1 and Σ_2 be based on two algebras $\mathcal{B}_1 \stackrel{\text{def}}{=} \mathcal{B}_{(\mathsf{Vars}_1,\mathsf{Acts}_1)}$ and $\mathcal{B}_2 \stackrel{\text{def}}{=} \mathcal{B}_{(\mathsf{Vars}_2,\mathsf{Acts}_2)}$ respectively and let $\mathcal{B} \stackrel{\text{def}}{=} \mathcal{B}_{(\mathsf{Vars}_1 \cup \mathsf{Vars}_2, \mathsf{Acts}_1 \cup \mathsf{Acts}_2)}$.

Furthermore, let

$$st_i \stackrel{\text{def}}{=} \{(\xi, \alpha, \xi') \in \mathcal{B}_i \mid \xi(x) = \xi'(x') \text{ for all } x \in \mathsf{Vars}_i \text{ and } \alpha(A) = \perp \text{ for all } A \in \mathsf{Acts}_i\} .$$

Then the canonical embeddings $\mathcal{B}_i \hookrightarrow \mathcal{B}$ map st_i to

$$\varphi_i(st_i) \stackrel{\text{def}}{=} \{(\xi, \alpha, \xi') \in \mathcal{B} \mid \xi(x) = \xi'(x') \text{ for all } x \in \mathsf{Vars}_i \text{ and } \alpha(A) = \perp \text{ for all } A \in \mathsf{Acts}_i\} .$$

Thus

$$\begin{aligned} st &\stackrel{\text{def}}{=} \varphi_1(st_1) \cap \varphi_2(st_2) \\ &= \{(\xi, \alpha, \xi') \in \mathcal{B} \mid \xi(x) = \xi'(x') \text{ for all } x \in \mathsf{Vars}_1 \cup \mathsf{Vars}_2 \text{ and} \\ &\qquad \alpha(A) = \perp \text{ for all } A \in \mathsf{Acts}_1 \cup \mathsf{Acts}_2\}. \end{aligned}$$

Besides, if $\mathsf{Vars}_1 \cap \mathsf{Vars}_2 = \mathsf{Acts}_1 \cap \mathsf{Acts}_2 = \emptyset$ then $0 < \varphi_1(st_1) \cap \varphi_1(M_1(e_1)) \cap \varphi_2(st_2) \cap \varphi_2(M_2(e_2))$ whenever $0 < \varphi_1(st_1) \cap \varphi_1(M_1(e_1))$ and $0 < \varphi_2(st_2) \cap \varphi_2(M_2(e_2))$ because $\varphi_i(\mathcal{B}_i)$ are independent subalgebras of $\mathcal{B}$. That is, [St_Independence], and therefore the composition of Σ_1 and Σ_2 is well-defined.

Also note that $0 < st_i \cap M_i(e_i)$ **iff** $0 < \varphi_i(st_i \cap M_i(e_i)) = \varphi_i(st_i) \cap \varphi_i(M_i(e_i))$ because φ is injective. Thus, [St_Independence] guarantees that two stutter steps in the original algebras $\mathcal{B}_1$ and $\mathcal{B}_2$ result in a stutter step in the composition.

Note, that this construction also generalizes to situations where $\mathcal{B} = \bigotimes_{i \in I} \mathcal{B}_i$ for a finite set I.

□

Similarly,

Definition 4.2.23 (Projection)

Let $\Sigma = (V, I, \mathcal{P}(X \times Y), st, M, \mathcal{F})$ be a BTS *over $\mathcal{P}(X \times Y)$ and $\big|_X : \mathcal{P}(X \times Y) \twoheadrightarrow \mathcal{P}(X)$ the canonical projection.*

Then the projection of Σ to $\mathcal{P}(X)$ is the BTS $\Sigma\big|_X \stackrel{def}{=} (V, I, \mathcal{P}(X), st\big|_X, M\big|_X, \mathcal{F}\big|_X)$ *with* $\mathcal{F}\big|_X \stackrel{def}{=} \{\mathsf{F}(E, \tilde{L}) \mid \mathsf{F}(E, L) \in \mathcal{F}$ *and* $\tilde{L}(e) = L(e)\big|_X \}$.

$\Sigma\big|_X$ is well-defined: Due to Lemma 4.1.18 (7), $\big|_X$ maps any non-zero element of $\mathcal{P}(X \times Y)$ to some non-zero element of $\mathcal{P}(X)$.

Lemma 4.2.24

1. $\mathbf{Runs}(\Sigma) = \mathbf{Runs}(\Sigma\big|_X)$
2. $\mathbf{Traces}(\Sigma\big|_X) = \mathbf{Traces}(\Sigma)\big|_X$
3. *If* $\mathbf{Traces}(\Sigma) \neq \emptyset$ *then* $\mathbf{Traces}(\Sigma\big|_X) \neq \emptyset$

Proof:

1. Immediate from Lemma 4.1.18 (7).
2. For this part, it is assumed that there are only Buechi acceptance conditions. For fairness conditions the proof is similar.

   $(x_0, x_1, \dots) \in \mathbf{Traces}(\Sigma\big|_X)$
   iff exists $(v_0, v_1, \dots) \in \mathbf{Runs}(\Sigma\big|_X)$ such that
   [1] $\{x_i\} \subseteq M(v_i, v_{i+1})\big|_X$ for all i
   [2] infinitely often $\{x_i\} \subseteq L(v_i, v_{i+1})\big|_X$ and $(v_i, v_{i+1}) \in E$ for all $\mathsf{Büchi}(E, L) \in \mathcal{F}_X$
   iff exists $(v_0, v_1, \dots) \in \mathbf{Runs}(\Sigma)$ such that
   [1] $\{x_i\} \subseteq M(v_i, v_{i+1})\big|_X$ for all i
   [2] infinitely often $\{x_i\} \subseteq L(v_i, v_{i+1})\big|_X$ and $(v_i, v_{i+1}) \in E$ for all $\mathsf{Büchi}(E, L) \in \mathcal{F}_X$
   iff exists $(v_0, v_1, \dots) \in \mathbf{Runs}(\Sigma)$ and exists $(y_0, y_1, \dots) \in Y^\omega$ such that
   [1] $\{(x_i, y_i)\} \subseteq M(v_i, v_{i+1})$ for all i
   [2] infinitely often $\{(x_i, y_i)\} \subseteq L(v_i, v_{i+1})$ and $(v_i, v_{i+1}) \in E$ for all $\mathsf{Büchi}(E, L) \in \mathcal{F}_X$
   iff exists $(y_0, y_1, \dots) \in Y^\omega$ such that $(\ (x_0, y_0), (x_1, y_1), \dots\ ) \in \mathbf{Traces}(\Sigma)$
   iff $(x_0, x_1, \dots) \in \mathbf{Traces}(\Sigma)\big|_X$
3. Immediately from Lemma 4.1.18 (8).

4.2.8 Abstraction

Whereas composition is needed to describe large specifications, a notion of refinement is necessary to develop specifications in a stepwise manner that guarantees that properties that once hold for a specification continue to hold in later development phases. The other way round, one may be interested in finding an abstraction for a given specification in such a way that firstly certain properties that hold for the abstraction also holds for the original specification and secondly the desired properties are easier to be checked for the abstraction.
As in case of composition that can be defined as trace intersection in a suitable Boolean algebra, refinement can be defined semantically as trace inclusion given a common Boolean algebra:

Definition 4.2.25 (Abstraction)

Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be complete, atomic Boolean algebras and $\Sigma_1 = (V_1, I_1, \mathcal{B}_1, st_1, M_1, \mathcal{F}_1)$ and $\Sigma_2 = (V_2, I_2, \mathcal{B}_2, st_2, M_2, \mathcal{F}_2)$ be Boolean Transition Systems.

Σ_2 is called an abstraction of Σ_1 and Σ_1 is called a refinement of Σ_2, written $\Sigma_1 \leq \Sigma_2$, **iff** *there exist a Boolean algebra $\mathcal{B}$ and monomorphisms $\varphi_i : \mathcal{B}_i \hookrightarrow \mathcal{B}$ such that* **Traces**$(\varphi_1\Sigma_1) \subseteq$ **Traces**$(\varphi_2\Sigma_2)$.

Obviously, $\Sigma_1 [\![\Sigma_2 \leq \Sigma_1$ and $\Sigma_1 [\![\Sigma_2 \leq \Sigma_2$ (as monomorphisms take the identity and φ_i).

The drawback of this semantical definition of refinement is, that it is hard to check. The following theorem presents some sufficient "structural" conditions that allow to deduce trace inclusion. In fact, it is not even necessary to find monomorphisms, as the theorem directly relates the Boolean algebras $\mathcal{B}_1$ and $\mathcal{B}_2$.

Theorem 4.2.26 (Abstraction Theorem)

Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be complete, atomic Boolean algebras and $\Sigma_1 = (V_1, I_1, \mathcal{B}_1, st_1, M_1, \mathcal{F}_1)$ and $\Sigma_2 = (V_2, I_2, \mathcal{B}_2, st_2, M_2, \mathcal{F}_2)$ be Boolean Transition Systems.

If there is a function Φ mapping

(1) states $v_1 \in V_1$ to states $v_2 \in V_2$,
(2) initial states $v_1 \in I_1$ to initial states $v_2 \in I_2$,
(3) atoms $s_1 \in S(\mathcal{B}_1)$ to atoms $s_2 \in S(\mathcal{B}_2)$ and arbitrary elements $b_1 \in \mathcal{B}_1$ to $\Phi(b_1) = \bigvee_{s \leq b_1} \Phi(s)$,

and such that

$$(4)\ M_2(v_2, w_2) \geq \bigvee_{\{(v_1,w_1) \in V_1 \times V_1 \mid \Phi(v_1)=v_2 \text{ and } \Phi(w_1)=w_2\}} \Phi(M_1(v_1, w_1)) ,$$

and if, furthermore, for any acceptance condition $\mathsf{F}(E_2, L_2) \in \mathcal{F}_2$ there is an acceptance condition $\mathsf{F}(E_1, L_1) \in \mathcal{F}_1$ such that

$$(5)\ E_2 \subseteq \{ (v_2, w_2) \in V_2 \times V_2 \mid \text{exists } (v_1, w_1) \in E_1 \text{ such that } \Phi(v_1) = v_2 \text{ and } \Phi(w_1) = w_2 \}$$

and

$$L_2(v_2, w_2) \geq \bigvee_{\{(v_1,w_1) \in E_1 \mid \Phi(v_1)=v_2 \text{ and } \Phi(w_1)=w_2\}} \Phi(L_1(v_1, w_1)).$$

*then $\Phi($**Traces**$(\Sigma_1)) \subseteq$ **Traces**(Σ_2).*

If furthermore Φ defines a Boolean algebra monomorphism $\varphi_1 : \mathcal{B}_1 \to \mathcal{B}_2$ then $\Sigma_1 \leq \Sigma_2$ by choosing $\varphi_2 \stackrel{def}{=} Id$.

Proof:

Let $(s_0, s_1, \dots) \in \mathbf{Traces}(\Sigma_1)$. Thus there exists a sequence $(v_0, v_1, \dots) \in \mathbf{Runs}(\Sigma_1)$. Then $(\Phi(v_0), \Phi(v_1), \dots) \in \mathbf{Runs}(\Sigma_2)$: [Run1] and [Run3] follow from (1) and (5) respectively. Furthermore, from $0 \neq M_1(v_1, w_1)$, it follows that $0 \neq \Phi(M_1(v_1, w_1)) \leq \bigvee_{\{(v,w) \in V_1 \times V_1 \mid \Phi(v)=\Phi(v_1) \text{ and } \Phi(w)=\Phi(w_1)\}} \Phi(M_1(v, w)) \leq M_2(\Phi(v_1), \Phi(w_1))$ and therefore [Run2].

It remains to show [Tr1] and [Tr2] for the sequence $(\Phi(s_0), \Phi(s_1), \dots) \in S(\mathcal{B}_2)^\omega$:
[Tr1] follows because

$$\begin{aligned}\Phi(s_i) &\leq \bigvee_{s \leq M_1(v_i, v_{i+1})} \Phi(s) = \Phi(M_1(v_i, v_{i+1})) \\ &\leq \bigvee_{\{(v,w) \in V_1 \times V_1 \mid \Phi(v)=\Phi(v_i) \text{ and } \Phi(w)=\Phi(v_{i+1})\}} \Phi(M_1(v, w)) \leq M_2(\Phi(v_i), \Phi(v_{i+1})).\end{aligned}$$

The first $\leq$-relation holds because especially $s_i \leq M_1(v_i, v_{i+1})$.

A similar argument shows that further $\Phi(s_i) \leq L_2(\Phi(v_i), \Phi(v_{i+1}))$ whenever required in [Tr2], that is, whenever $s_i \leq L_1(v_i, v_{i+1})$. From this, one can deduce [Tr2].

■

REMARK 1. Φ as defined in (3) is not in general a Boolean algebra homomorphism. For example, let $\mathcal{B}_1 = \mathcal{P}(\{1,2,3\})$, $\mathcal{B}_2 = \mathcal{P}(\{A, B\})$, $\Phi(\{1\}) = \Phi(\{2\}) = \{A\}$, and $\Phi(\{3\}) = \{B\}$. Then, $\Phi(\{1,3\} \cap \{2,3\}) = \Phi(\{3\}) = \{B\} \subsetneqq \{A, B\} = \Phi(\{1,3\}) \cap \Phi(\{2,3\})$.

REMARK 2. If $\mathcal{B}_1 = \mathcal{P}(X \times Y)$ and $\mathcal{B}_2 = \mathcal{P}(X)$ then the canonical projection $\big|_X : \mathcal{P}(X \times Y) \twoheadrightarrow \mathcal{P}(X)$ fulfills condition (3) from above. From Lemma 4.1.18 (8), it follows that $\big|_X$ maps atoms to atoms. Furthermore, for arbitrary $T \subseteq X \times Y$

$$\Phi(T) = \bigcup_{(x,y) \in T} \Phi(\{(x, y)\}) = \bigcup_{(x,y) \in T} \{x\} = \{ x \in X \mid \exists_{y \in Y} (x, y) \in T \} = T\big|_X$$

Taking the identity function as Φ, it follows immediately that omitting acceptance conditions results in an abstraction:

Corollary 4.2.27
Let $\Sigma_1 = (V, I, \mathcal{B}, st, M, \mathcal{F}_1)$, $\Sigma_2 = (V, I, \mathcal{B}, st, M, \mathcal{F}_2)$ and $\mathcal{F}_2 \subseteq \mathcal{F}_1$. Then $\Sigma_1 \leq \Sigma_2$.

As a second example, consider two BTS Σ_1 and Σ_2 that are independent in the sense that $\mathcal{B}_1 = \mathcal{P}(X)$ and $\mathcal{B}_2 = \mathcal{P}(Y)$ are embedded canonically into some labeling algebra $\mathcal{B} = \mathcal{P}(X \times Y \times Z)$. In this situation, the function Φ required in Theorem 4.2.26 is defined as follows:

$\Phi((v_1, v_2)) \stackrel{\text{def}}{=} v_1$ for all states $(v_1, v_2) \in V_1 \times V_2$,
$\Phi((v_1, v_2)) \stackrel{\text{def}}{=} v_1$ for all initial states $(v_1, v_2) \in I_1 \times I_2$, and
$\Phi(\{(x, y, z)\}) \stackrel{\text{def}}{=} \{x\}$ for all atoms $\{(x, y, z)\} \in S(\mathcal{B})$.

Note, that Φ coincides with the canonical projection $\big|_X$ that maps subsets $T \subseteq X \times Y \times Z$ to $T\big|_X = \{x \in X \mid \exists_{(y,z) \in Y \times Z} (x, y, z) \in T\} = \bigcup_{\{(x,y,z)\} \subseteq T} \{x\}$.

Chapter 5

A Programming Notation

Specifications could be given purely on the logical level (as in TLA) or even on the automata level only (as in discrete-event systems or statecharts). However, there are good reasons for introducing a specification language that is somewhat "higher" than temporal logic or transition systems:

- Specifications get more structure by distinguishing between input and output, blocks and modules, or guarded and triggered transitions for example.
- The use of parameters, macros and short hand notations (like If- Then- Else) allows for concise specifications.
- Specifications are built up hierarchically. The base is given by blocks which form the smallest syntactic unit with context independent meaning. Modules then are sets of blocks sharing the same set of declarations. On the system level, finally, an interpretation and a valuation of the parameters of the system are fixed, and the modules are interconnected in essence by use of renaming.
- It is possible to specialize the specification language allowing certain adapted "programming styles". These may, for example, ease verification by restricting language constructs (for example, the restriction to finite data types allows the use of model checking techniques, i.e. decision procedures). Other restrictions may ease the implementation of specifications. Typical RPC or client server applications, for example, use the "acknowledge protocol" from the introductory example to call "lower level" modules. This protocol allows to replace the synchronous communication by asynchronous one and therefore the implementation of TLT modules as set of UNIX processes running on a LAN.

In the following, one such specification style will be presented. It is specialized to deal with distributed reactive systems and eases assumption-commitment style proofs by allowing (in some sense even forcing) to annotate the specifications. Each building block of this specification language will come along with syntactic as well as logical verification conditions that together guarantee the realizability of the specification (that is, the existence of traces).

Other specification styles (further annotations) for fault-tolerance, real-time, or performance evaluation are conceivable. One style, specialized to deal with client server architectures, is described in [Bar97].

Remark. To increase readability, syntactic entities (variables, functions, etc. but also keywords of the specification language) will be printed using sans serif fonts if they occur in normal text. In specifications, only the keywords are printed using sans serif fonts. Furthermore, most syntactic entities are defined as tuples $\langle \mathsf{Id}, \mathsf{SubDef}_1, \ldots, \mathsf{SubDef}_n \rangle$ with Id uniquely identifying the entity. Then the notation $\mathsf{SubDef}_i(\mathsf{Id})$ is used instead of simply SubDef_i whenever the context is not obvious.

5.1 Types

In section 3.1.1, variables and actions were given a unique sort. In the context of programming languages, one usually talks of types instead of sorts. Some typical "build-in" types are Boolean, Bit, () (for the unit type), String, Natural, Integer and Real. Another basic type is [M..N] representing an integer interval. Thus, M and N must be integer constants or system parameters denoting the lower and upper bound of the interval. Finite sets of constants are described as usual by enumerating the elements separated by commas and enclosed in curly brackets.

Besides these basic types, it is possible to form complex types from given (basic or complex) types. Complex types may be defined as tuples, unions, set differences, arrays or lists:

$\mathsf{Type}_1 \times \mathsf{Type}_2$	Cartesian product of Type_1 and Type_2
$\mathsf{Type}_1 \cup \mathsf{Type}_2$	union of elements from Type_1 and Type_2
$\mathsf{Type}_1 - \mathsf{Type}_2$	set of all elements from Type_1 not in Type_2
Array [M ... N]Of Type_1	function $[\mathsf{M} \ldots \mathsf{N}] \rightarrow \mathsf{Type}_1$
List Of Type_1	list with elements of type Type_1

In TLT specifications, user-defined types T_i are defined in the Types-section as follows:

```
Types
  T1  :=  type_definition1

  T2  :=  type_definition2

      ⋮
```

For example,

Motor_States := { stopped, running, defect }

defines a type Motor_States whose values are taken from a set of constants indicating possible states of a motor.

Index := [1..N]

defines a type Index whose values range from 1 to N, the latter being a parameter of type Natural or Integer.

5.2 Parameters

Specifications in TLT may be parameterized in order to avoid an inflation of virtually identical specifications. Examples for parameterized (or generic) specifications are a lift specification for "N" floors or a mutual exclusion algorithm for "N" processes. Formally, a parameter is nothing but a specification variable:

Definition 5.2.1 (Parameters)
Tuples $<$ id, type $>$ *or* $<$ id, type, expr $>$ *are called parameter definition* **iff**

- $\mathsf{id} \in \Gamma_{\mathsf{S}}$ *is a unique identifier,*
- $\mathsf{type} = \mathrm{sort}(\mathsf{id}) \ \in \Gamma_{\mathsf{Sort}}$, *and*
- expr *is a term.*

The parameters of a specification are listed in the Parameters-section as follows:

Parameters
Param_1 : type_1 $:= \mathrm{expr}_1$
Param_2 : type_2
⋮

Parameterized specifications are presented in Section 5.10.

5.3 Declarations

For purposes of compositional specification, program (flexible) variables and actions are now further enriched with an environment class. The environment class determines the 'ownership' and 'scope of visibility' of variables and actions in a modular system.

The intuition behind the variable classes will be as follows: Local variables belong to a particular module and are not visible to the outside. Write variables belong to the module which declares them as write; they are visible to the environment, but they may not be modified by it. Read variables are imported from the environment; their values can be read but not modified locally. Spec variables get used in quantifications, as formal parameters of actions, and as array indices; and finally, History variables record the history of visible events between a module and its environment; they are auxiliary variables in the sense that their values are completely determined by the history of visible events.

Remark. An environment class for Global variables is not provided. Our own experience has discouraged their use, because the question of ownership, the consequences for composition and difficulties with implementation make global variables troublesome. An equivalent but more elegant solution is to implement global variables as separate modules, in the style of abstract data types with appropriate access methods (see Page 128 for an example).

Input and output actions intuitively provide a user with value-passing synchronous or rendezvous communication, which is performed on a 1-1 or 1-many basis per 'communication channel'. The action classes can be interpreted as follows: In actions are actions under the control of the environment, and Out actions are under control of a given module. In a composed system,

input and output actions with the same identifier match each other, but only one module may declare an action as Out action. In order to hide actions for refinement purposes, one has to add a third class of so-called Internal actions. These might also be used as internal events to synchronize among the blocks of a module. However, for this latter purpose one might also use Out actions that are simply "ignored" by other modules.

Definition 5.3.1 (Declarations)
A triple $<$ id, type, env_class $>$ *is called declaration* **iff**

- $\mathsf{id} \in \Gamma_{\mathsf{Var}} \cup \Gamma_{\mathsf{Act}}$ *is a unique identifier*
- $\mathsf{type} = \mathrm{sort}(\mathsf{id}) \ \in \Gamma_{\mathsf{Sort}}$
- env_class *is an element from* {Local, Write, Read, Spec, History, Out, In, Internal} *such that*
  $\mathsf{env\_class} \in \{\mathsf{Local}, \mathsf{Write}, \mathsf{Read}, \mathsf{Spec}, \mathsf{History}\}$ *for all* $\mathsf{id} \in \Gamma_{\mathsf{Var}}$
  $\mathsf{env\_class} \in \{\mathsf{Out}, \mathsf{In}, \mathsf{Internal}\}$ *for all* $\mathsf{id} \in \Gamma_{\mathsf{Act}}$

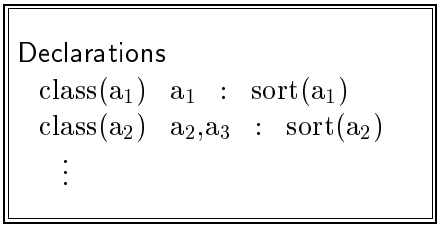
The function sort(id) *is used to refer to the type and similarly a function* class(id) *is used to refer to the environment class of* id. *Whenever* class(id) $\in$ {Local, Write, Read, History} *then a primed copy of* id *with identical type and environment class is defined implicitly.*

Let Decl *denote a set of declarations. Then*

(1) Vars *denotes the set of declared program variables (that is, all variables* v *such that* class(v) $\neq$ Spec *. The following subsets of* Vars *are distinguished:*
- vVars, *the set of variables* v *with* class(v) $\in$ {Write, Read} *(the visible variables),*
- hVars, *the set of variables* v *with* class(v) $\in$ {Local, History} *(the hidden variables), and*
- lVars, *the set of variables* v *with* class(v) = Local *(the local variables).*

(1) Acts *denotes the set of declared actions. Again, two subsets are distinguished:*
- vActs, *the set of actions* A *with* class(A) $\in$ {Out, In} *(the visible actions),*
- hActs, *the set of actions* A *with* class(A) = Internal *(the hidden (or internal) actions).*

The declarations of variables and actions $\mathsf{a_i}$ are listed in the Declarations-section as follows:



For example,

In Call : ()

declares a signal Call controlled by the environment. Variables or actions with common environment class and sort may be summarized to one declaration, e.g.

In Call_1, Call_2 : ()

5.4 Abbreviations

Abbreviations or macros are a necessary means to obtain readable specifications. In TLT, abbreviations may be introduced for arbitrary terms, formulas or commands (see Section 5.6.2 below). For example,

max_length := 65536

introduces a symbolic name for an integer constant. Abbreviations get resolved in a preprocessing step by syntactically replacing the identifiers on the left by the terms, formulas or commands on the right.

Definition 5.4.1 (Abbreviation)
A tuple $<$ id, expr $>$ *is called an abbreviation* **iff**

- id *is a unique identifier, and*
- expr *is either a term, a formula, or a command*

In the programming notation, the macros are listed in the Abbreviations-section:

Abbreviations
  Abbrev_1 := expr_1
  Abbrev_2 := expr_2
  ⋮

5.5 Initial Values

Initial values may be specified in two ways:

Firstly, it is possible to specify an initial predicate init in the initial section:

Initially init

Formally, an initial predicate init is a state predicate such that class(x) $\in$ {Write, Local, History} for all x $\in$ FV(init). Any valuation satisfying the initial predicate may serve as an initial state. If no initial predicate is specified, then true is assumed.

Secondly, it is allowed to extend the declaration of Local, Write, and History variables x by specifying a term init_expr of the same type as the variable declared:

class(x) x : sort(x) Init init_expr

with sort(x) = sort(init_expr). In this case, $\wedge$ x = init_expr is appended to the initial predicate. Thus, wlog. there is only one initial predicate.

5.6 Transitions

Transitions on the logical level are given as transition predicates. On the level of the specification language, transition predicates still form the basis of transitions. However, additional structure

is added.

Firstly, events are introduced as a special kind of transition predicates.

5.6.1 Events

Informally, an event describes a non stuttering step, that is, a transition whose occurrence is "observable" and which usually results in a state change. In-events are used in modules to describe "stimuli" of the environment. For example, consider an action Call whose occurrences are to be counted. This would be written as

$$\mathsf{Call} \Rightarrow\triangleright \mathsf{number\_of\_calls}' = \mathsf{number\_of\_calls} + 1$$

and may be read as "whenever a Call occurs, the counter number_of_calls is increased by one". Similarly,

$$\mathsf{Call} \wedge \neg\mathsf{Return} \Rightarrow\triangleright \mathsf{pending\_calls}' = \mathsf{pending\_calls} + 1$$

has to be read as "whenever a Call but no Return occurs, the counter pending_calls is increased by one". But what about

$$\neg\mathsf{Call} \Rightarrow\triangleright \cdots \quad ?$$

The problem in the last example arises due to the fact that modules are allowed to stutter. Thus, if Call is an input action, it can not be decided whether another module explicitly made a non stuttering step labelled by ¬Call or whether it simply stuttered. The same problem arises for predicates, e.g. "if the value for the temperature increases then send out a warning", noted by

$$\mathsf{temp}' > \mathsf{temp} \Rightarrow\triangleright \mathsf{Warning}$$

is executable, whereas

$$\mathsf{temp}' \geq \mathsf{temp} \Rightarrow\triangleright \mathsf{Warning}$$

does not make sense in a context where stuttering is allowed. Of course, this is not simply due to the fact that $\geq$ is reflexive, whereas $>$ is not (take $\mathsf{temp}' > \mathsf{temp} - 1$ and $\mathsf{temp}' \geq \mathsf{temp} + 1$).

One easy solution is to define events as transition predicates p that satisfy the additional constraint that

(*) there is no valuation (ξ, τ, ξ') with $\xi'(x') = \xi(x)$ for all $x \in \Gamma_V$ such that $[\![p]\!]_{\mathcal{I},\gamma}^{(\xi,\tau,\xi')}$.

Here and in the sequel, τ denotes the valuation that assigns $\perp$ to all actions. Note, that (*) implies that p is not a state predicate. However, such semantical definitions are hard to validate. Furthermore, synchronization would be based on actions and variables. Thus, in this thesis, attention will be restricted to events built up of actions only. This results in a purely syntactical definition:

Definition 5.6.1 (Events)

Events are defined with regard to a set Decl *of declarations and all actions occurring in an event have to be declared in* Decl. *Given such a set, atomic events are:*

- *actions of form* $\mathsf{A(t)}$, *such that* $\mathsf{FV(A(t))} \subseteq \Gamma_\mathsf{S}$, *or*
- *actions of form* $\exists_{\mathsf{c_1},\ldots\mathsf{c_k}}\mathsf{A(t)}$ *such that* $\mathsf{FV}(\exists_{\mathsf{c_1},\ldots\mathsf{c_k}}\mathsf{A(t)}) \subseteq \Gamma_\mathsf{S}$

Events then are defined as

$$ev = \bigvee_{i \in I} (\bigwedge_{j \in J_i} at\_ev_j \ \wedge \bigwedge_{k \in K_i} \neg at\_ev_k)$$

where $\mathsf{at\_ev}_j$ *and* $\mathsf{at\_ev}_k$ *are atomic events, and* $I, J_i \neq \emptyset$ *for all* i.

An event ev *is called* In*-event iff* class(A) = In *for all* A ∈ FAct(ev).
An event ev *is called* Out*-event iff* class(A) ∈ {Out, Internal} *for all* A ∈ FAct(ev).

$\exists_{\mathsf{c}}\mathsf{A}(\mathsf{c})$ is used to express "some value is sent on channel A". In the sequel, $\exists_{\mathsf{c}}\mathsf{A}(\mathsf{c})$ will be abbreviated by A. This can not be confused with the abbreviation $\mathsf{A} \stackrel{\text{def}}{=} \mathsf{A}(\surd)$ introduced for signals, because $[\![A(\surd)]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$ **iff** $[\![\exists_c A(c)]\!]^{(\zeta,\alpha,\zeta')}_{\mathcal{I},\gamma}$ for a specification variable c of sort ().

Lemma 5.6.2
Events fulfill condition (*) from above, i.e. there is no valuation $\beta = (\xi, \tau, \xi')$ *with* $\xi'(x') = \xi(x)$ *for all* $x \in \Gamma_V$ *such that* $[\![ev]\!]^{\beta}_{\mathcal{I},\gamma}$.

Proof
Suppose $[\![ev]\!]^{\beta}_{\mathcal{I},\gamma}$ for some valuation $\beta = (\xi, \tau, \xi')$. That is, $[\![\bigwedge_{j \in J_i} at\_ev_j \ \wedge \bigwedge_{k \in K_i} \neg at\_ev_k]\!]^{\beta}_{\mathcal{I},\gamma}$ for some $i \in I$. Especially, $[\![\bigwedge_{j \in J_i} at\_ev_j]\!]^{\beta}_{\mathcal{I},\gamma}$ for some $i \in I$. As $J_i \neq \emptyset$ there exists an atomic event at_ev_j such that $[\![at\_ev_j]\!]^{\beta}_{\mathcal{I},\gamma}$. Two cases are possible:

- at_ev_j is an action $A(t)$. Then
  $[\![A(t)]\!]^{\beta}_{\mathcal{I},\gamma}$ **iff** $\beta(A) = [\![t]\!]^{\beta}_{\mathcal{I},\gamma}$ which contradicts $\beta(A) = \tau(A) = \bot$.
- at_ev_j is an action $\exists_{c_1,\ldots,c_k} A(t)$. Then
  $[\![\exists_{c_1,\ldots,c_k} A(t)]\!]^{\beta}_{\mathcal{I},\gamma}$ **iff** $[\![A(t)]\!]^{\beta}_{\mathcal{I},\gamma[c_1 \leftarrow d_1 \ldots c_k \leftarrow d_k]}$ for some $d_i \in \mathcal{I}(\mathrm{sort}(c_i))$
  iff $\beta(A) = [\![t]\!]^{\beta}_{\mathcal{I},\gamma[c_1 \leftarrow d_1, \ldots c_k \leftarrow d_k]}$
  which contradicts $\beta(A) = \tau(A) = \bot$.

■

From this proof, it follows immediately that the evaluation of events does not depend on ξ or ξ'. Therefore, Lemma 5.6.2 can be strengthened to

Corollary 5.6.3
If $\beta(A) = \bot$ *for all* $A \in \mathsf{FAct}(ev)$ *then* $[\![\neg ev]\!]^{\beta}_{\mathcal{I},\gamma}$.

5.6.2 Commands

Next, some structure is added to transition predicates, resulting in commands.
Remark: As in the case of formulas, the syntax of transitions will be defined inductively. This style allows brief definitions by avoiding to talk about priorities of operators etc. As in the case of formulas, a "flat" representation of a command is typically ambiguous. Such ambiguities are resolved by use of parenthesis.

Before introducing commands, an abbreviation is presented that will be used frequently in the sequel. Informally, Stutter describes that nothing happens with regard to its argument. If the argument is a variable, this means that its value remains unchanged. If the argument is an action, this means that it does not happen. Formally,

Definition 5.6.4 (Stutter)

For an action or a program variable a, $\mathsf{Stutter}(a) \stackrel{def}{=} \begin{cases} a' = a & \text{, for } a \in \Gamma_V \\ \neg\exists_c\ a(c) & \text{, for } a \in \Gamma_{Act} \end{cases}$

For a set $\mathcal{A}$ *of actions and program variables,* $\mathsf{Stutter}(\mathcal{A}) \stackrel{def}{=} \bigwedge_{a \in \mathcal{A}} Stutter(a)$

Now, together with the syntax of commands cmd, their control set Ctr(cmd) containing the variables and actions controlled by the transition, and a function Trl, mapping commands to transition predicates, are defined.

Definition 5.6.5 (Commands and their Control Set)
Commands are defined with regard to a set Decl *of declarations and all variables and actions occurring in a command have to be declared in* Decl. *Given such a set,*

1. *transition predicates* p *are commands if* class(A)$\in$ {Out,Internal} *for all actions* A *occurring in* p *and* class(x′)$\neq$Read *for all primed variables occurring (free) in* p.

   $\mathrm{Ctr}(p)$={x | class(x)$\in$ {Local,Write,History} *and* x′ *occurs in* p} $\cup$
   {A | class(A)$\in$ {Out,Internal} *and* A *occurs in* p}

   $\mathrm{Trl}(p)=p$

2. $\mathsf{cmd}_1 \parallel \mathsf{cmd}_2$ *is a command for commands* cmd_1 *and* cmd_2 [1]

   $\mathrm{Ctr}(\mathsf{cmd}_1 \parallel \mathsf{cmd}_2) = \mathrm{Ctr}(\mathsf{cmd}_1) \cup \mathrm{Ctr}(\mathsf{cmd}_2)$

   $\mathrm{Trl}(\mathsf{cmd}_1 \parallel \mathsf{cmd}_2) = \mathrm{Trl}(\mathsf{cmd}_1) \wedge \mathrm{Trl}(\mathsf{cmd}_2)$

3. If p Then cmd_1 Else cmd_2 *is a command for commands* cmd_1 *and* cmd_2 *and a transition predicate* p *such that* FAct$(p)=\emptyset$ *and* class(x′)$\neq$Read *for all primed variables occurring (free) in* p, cmd_1 *or* cmd_2.

   Ctr(If p Then cmd_1 Else cmd_2) $= \mathrm{Ctr}(\mathsf{cmd}_1) \cup \mathrm{Ctr}(\mathsf{cmd}_2)$

   Trl(If p Then cmd_1 Else cmd_2) $=$ $p \wedge \mathrm{Trl}(\mathsf{cmd}_1) \wedge \mathsf{Stutter}(\mathrm{Ctr}(\mathsf{cmd}_2) - \mathrm{Ctr}(\mathsf{cmd}_1)) \vee \neg\, p \wedge \mathrm{Trl}(\mathsf{cmd}_2) \wedge \mathsf{Stutter}(\mathrm{Ctr}(\mathsf{cmd}_1) - \mathrm{Ctr}(\mathsf{cmd}_2))$

   As usual, Else true *may be omitted.*

4. Case
   ‖ p_1:cmd_1
   ⋮
   ‖ p_k:cmd_k
   End
   is a command for commands cmd_i *and transition predicates* p_i *such that* FAct$(p_i)=\emptyset$ *and* class(x′)$\neq$Read *for all primed variables occurring (free) in one of the* p_i *or* cmd_i.

$$\mathrm{Ctr}\left(\begin{array}{l}\textsf{Case}\\ \quad \| \ p_1:\mathsf{cmd}_1\\ \quad \vdots\\ \quad \| \ p_k:\mathsf{cmd}_k\\ \textsf{End}\end{array}\right) = \bigcup_{1 \le i \le k} \mathrm{Ctr}(cmd_i) \quad ,$$

$$\mathrm{Trl}\left(\begin{array}{l}\mathsf{Case}\\ \quad [\!] \ p_1 : \mathsf{cmd}_1\\ \quad \vdots\\ \quad [\!] \ p_k : \mathsf{cmd}_k\\ \mathsf{End}\end{array}\right) = \bigvee_{1 \le \mathsf{i} \le \mathsf{k}} p_i \wedge \mathrm{Trl}(\mathsf{cmd_i}) \wedge \mathsf{Stutter}(\bigcup_{\mathsf{j} \neq \mathsf{i}} \mathrm{Ctr}(\mathsf{cmd_j}) \; - \; \mathrm{Ctr}(\mathsf{cmd_i}))$$

Furthermore, let $\mathrm{Ctr}_V(\mathsf{cmd}) \stackrel{def}{=} \mathrm{Ctr}(\mathsf{cmd}) \cap \Gamma_V$,
$\mathrm{Ctr}_{Act}(\mathsf{cmd}) \stackrel{def}{=} \mathrm{Ctr}(\mathsf{cmd}) \cap \Gamma_{Act}$, *and*
$\mathrm{Ctr}'(\mathsf{cmd}) \stackrel{def}{=} \{x' \in \Gamma_{V'} \mid x \in \mathrm{Ctr}(\mathsf{cmd})\} \; \cup \; \{A \in \Gamma_{Act} \mid A \in \mathrm{Ctr}(\mathsf{cmd})\}$.

REMARK 1. If p Then c_1 Else c_2 has the same translation as $\begin{array}{l}\mathsf{Case}\\ \quad p : \mathsf{c}_1\\ \quad \neg p : \mathsf{c}_2\\ \mathsf{End}\end{array}$. For example:

$$\mathrm{Trl}\left(\begin{array}{r}\mathsf{If\ x{=}y\ Then\ x'{=}x{+}1}\\ \mathsf{Else\ y'{=}y{+}1}\end{array}\right) \equiv \mathrm{Trl}\left(\begin{array}{l}\mathsf{Case}\\ \quad [\!] \ \mathsf{x{=}y : x'{=}x{+}1}\\ \quad [\!] \ \mathsf{x{\neq}y : y'{=}y{+}1}\\ \mathsf{End}\end{array}\right) \equiv \begin{array}{l}\mathsf{x} = \mathsf{y} \wedge \mathsf{x}' = \mathsf{x}+1 \wedge \mathsf{y}' = \mathsf{y}\\ \vee \; \mathsf{x} \neq \mathsf{y} \wedge \mathsf{y}' = \mathsf{y}+1 \wedge \mathsf{x}' = \mathsf{x}\end{array}$$

where $\equiv$ denotes syntactical equality.

5.6.3 Triggered Transitions

Triggered transitions in essence consist of an event and a command. The intuition is that whenever the event happens, then the corresponding command is performed. Triggered transitions serve mainly two purposes:

1. They are used to process input actions: typically an input event would trigger an update of variables or trigger a suitable output action of the module.

2. They are used to set the values of history variables, which record the history of visible events; history variables play an important role in modeling the behaviour of the environment as part of the assumption-commitment style of specification in TLT.

In Chapter 4 the notions "being input enabled" and "at most delaying" were defined in order to guarantee non-contradicting ("deadlock free") composition. Here, by default, triggered transitions may not block their environment (i.e. they are input enabled). This may be weakened in two ways. Firstly, by delaying the occurrence of the event until a state predicate is satisfied. This results in the proof obligation that after a finite number of local steps this predicate will be satisfied. Secondly, assumptions made about the behaviour of the environment may be formulated. Again, these assumptions are state predicates. Here, the intuition is that the event happens only if the state predicate is satisfied. This results in a proof obligation for the environment.

Definition 5.6.6 (Triggered Transitions)
Triggered Transitions are defined with regard to a set Decl *of declarations and all variables and actions occurring in the transition have to be declared in* Decl. *Given such a set, a tuple* $<$ tt, s, assume, delay, ev, cmd $>$ *is called triggered transition* **iff**

[1] To ease implementation one may require $\mathrm{Ctr}(\mathsf{cmd}_1) \cap \mathrm{Ctr}(\mathsf{cmd}_2) = \emptyset$ and thereby forbid, for example, $\mathsf{x}' < 3 \parallel \mathsf{x}' > 0$.

- tt *is an optional name for reference purposes,*
- s *is a possibly empty list of specification variables,*
- assume *is a state predicate with* $\mathrm{class}(v) \neq \mathsf{Local}$ *for all* $v \in \mathsf{FV}(\mathsf{assume})$,[2]
- delay *is a state predicate with* $\mathrm{class}(v) \in \{\mathsf{History}, \mathsf{Local}, \mathsf{Spec}\}$ *for all* $v \in \mathsf{FV}(\mathsf{delay})$
- ev *is either an* In*-event or an* Out*-event,*
- cmd *is a command defined with regard to* Decl*, and*
- [TT] $[\![\forall_{\mathsf{s}} \; (\; \mathsf{assume} \wedge \mathsf{delay} \wedge \mathsf{ev} \Rightarrow \exists_{\mathrm{Ctr}'(\mathsf{cmd})} \; \mathrm{Trl}(\mathsf{cmd}) \;)]\!]_{\gamma,\mathcal{I}}$

The conjunction of the state predicates assume *and* delay *is called* guard *of a triggered transition.*

Further, it is required that delay=true *whenever* ev *is an* Out*-event. In this case, the assumption* assume *is also referred to as commitment (the module the transition belongs to commits itself to cause* ev *only if* assume *holds).*

Notation: $[\mathsf{tt}]_{\mathsf{s}} \; \{\mathsf{assume}\} \; \langle \mathsf{delay} \rangle \qquad \mathsf{ev} \Rightarrow\!\triangleright \mathsf{cmd}$

$\{\mathsf{true}\}$ and $\langle \mathsf{true} \rangle$ may be and usually are omitted.

The specification variables in s serve as formal parameters binding the values of actions occurring in the event. For example,

$[\![_c \; Call(c) \Rightarrow\!\triangleright pnd' = c$

means, that if a Call happens, then its value is bound to the specification variable c that implicitly is declared to be of the same type as Call.

Often, triggered transitions are used to update (history) variables. Thus, the following shorthand notation is used for triggered transitions:

$[\![\;\; A(h')$ instead of $[\![_c \; A(c) \Rightarrow\!\triangleright h' = c$

or

$[\![\;\; A(h') \Rightarrow\!\triangleright \mathrm{sth}$ instead of $[\![_c \; A(c) \Rightarrow\!\triangleright \mathrm{sth} \parallel h' = c$

More general, the "pure" syntax may be recovered by scanning ev for action symbols containing primed variables x' in their "parameter list", replacing these by fresh specification variables c, adding these to s, and adding either $\Rightarrow\!\triangleright x' = c$ or $\parallel x' = c$ to the right hand side depending on whether there already is a right hand side.

5.6.4 Guarded Transitions

Guarded transitions basically consist of a state predicate representing a guard (guard) and a command (cmd). Intuitively, this means that once in a state where the guard is *enabled* (i.e., evaluates to true), it is possible to execute the command cmd. The TLT semantics in fact allows any subset of enabled transitions to be executed simultaneously, as long as they do not contradict each other. In this sense, TLT has a *step* semantics and not an *interleaving* semantics.

[2] It is allowed that assume=false meaning that the corresponding ev never occurs.

The execution of an individual, enabled transition is of course just a special case. Furthermore, the implicit consistency criterion associated with each guarded transition only guarantees the executability of the transition itself, and not for arbitrary subsets.

Definition 5.6.7 (Guarded Transitions)
Guarded transitions are defined with regard to a set Decl *of declarations and all variables and actions occurring in a guarded transition have to be declared in* Decl. *Given such a set, a triple* $<$ gt, guard, cmd $>$ *is called guarded transition* **iff**

- gt *is an optional identifier for reference purposes,*
- guard *is a state predicate,*
- cmd *is a command defined with regard to* Decl, *and*
- [GT] $[\![\mathsf{guard} \Rightarrow \exists_{\mathrm{Ctr}'(\mathsf{cmd})}\ \mathrm{Trl}(\mathsf{cmd})]\!]_{\gamma,\mathcal{I}}$

Notation:

[gt] guard $\rightarrow$ cmd

5.6.5 Fairness Conditions

Intuitively, transitions describe one-step behavior only. To describe the ongoing behavior of a specification, infinite sequences of transitions are to be considered. As for the triggered transitions, there is no choice: they *must* be executed whenever the corresponding event occurs. On the other hand, for guarded transitions, there is too much freedom, because any subset of simultaneously enabled guarded transitions can be executed. Allowing any execution sequence (or, in terms of automata any paths) is too coarse in this situation. For example, it would be legal to never execute any guarded transition. Thus a kind of scheduling is necessary. However, the scheduling should neither be too restrictive nor too specialized. Fairness serves as such a general scheduler. Intuitively, it works simply by being fair" to all transitions. Or, in other words, it avoids favoring particular transitions in an unfair way.

Definition 5.6.8 (Fairness Conditions)
Fairness conditions are defined with regard to a set Decl *of declarations and all variables and actions occurring in a fairness condition have to be declared in* Decl. *Given such a set,* $<$ fair, F, fg, fcmd $>$ *is called fairness condition* **iff**

- fair *is an (optional) name for reference purposes,*
- F *is either* WF *or* SF,
- fg *is a state predicate, and*
- fcmd *is a command.*

Notation:

[fair] F(fg, fcmd)

Typically, fairness conditions are closely linked to a guarded transition. Therefore, [fair] F(gt) abbreviates [fair] F(guard, cmd) for some guarded transition $<$ gt, guard, cmd $>$.

5.7 Blocks

Blocks are the smallest unit with context independent meaning. They define automata or temporal logic formulas in such a way that

(1) they are executable (i.e., there exists at least one trace),
(2) the same translation of a block is used independently from the context, and
(3) under reasonable consistency criteria, the composition of blocks is still executable (i.e., care is taken that blocks can not contradict).

Definition 5.7.1 (Blocks)

A block $<$ Bl, Types, Params, Decls, Init, Transitions, Fair $>$ *consists of*

- Bl, *a unique identifier for reference purposes,*
- Types, *a set of type definitions as described in Section 5.1,*
- Params, *a set of parameter declarations as described in Section 5.2,*
- Decls, *a non-empty set of declarations as introduced in Definition 5.3.1,*
- Init, *a state predicate,*
- Transitions, *a finite set of transitions, and*
- Fair, *a finite set of fairness conditions.*

Writing all triggered transitions at the beginning, wlog, Transitions $=$
$\{ <$ tt_1, s_1, assume_1, delay_1, ev_1, cmd_1 $>, \ldots, <$ $\mathsf{tt}_{|\mathsf{tt}|}$, $\mathsf{s}_{|\mathsf{tt}|}$, $\mathsf{assume}_{|\mathsf{tt}|}$, $\mathsf{delay}_{|\mathsf{tt}|}$, $\mathsf{ev}_{|\mathsf{tt}|}$, $\mathsf{cmd}_{|\mathsf{tt}|}$ $>$,
$<$ $\mathsf{gt}_{|\mathsf{tt}|+1}$, $\mathsf{guard}_{|\mathsf{tt}|+1}$, $\mathsf{cmd}_{|\mathsf{tt}|+1}$ $>, \ldots, <$ $\mathsf{gt}_{|\mathsf{tt}|+|\mathsf{gt}|}$, $\mathsf{guard}_{|\mathsf{tt}|+|\mathsf{gt}|}$, $\mathsf{cmd}_{|\mathsf{tt}|+|\mathsf{gt}|}$ $> \}$.

Further, let $g_i \stackrel{def}{=} assume_i \wedge delay_i$ *for all* $1 \leq i \leq |tt|$.

If $\bigcup_{1 \leq i \leq |tt|+|gt|} \mathrm{Ctr}_V(\mathsf{cmd_i}) \neq \emptyset$, *then the set of controlled variables and actions is defined by*

$$\mathrm{Ctr}(\mathsf{block}) \stackrel{def}{=} \bigcup_{1 \leq i \leq |tt|+|gt|} \mathrm{Ctr}(\mathsf{cmd_i})$$

else

$$\mathrm{Ctr}(\mathsf{block}) \stackrel{def}{=} \{v\}$$

for a fresh variable $<$ v, (), Local $>$.

Finally, let $\mathrm{Ctr}_V(\mathsf{block}) \stackrel{def}{=} \mathrm{Ctr}(\mathsf{block}) \cap \Gamma_V$ *and* $\mathrm{Ctr}_{Act}(\mathsf{block}) \stackrel{def}{=} \mathrm{Ctr}(\mathsf{block}) \cap \Gamma_{Act}$.

With these definitions, the following consistency conditions are required of blocks:

- *All transitions are defined with regard to* Decls.
- *All* History *variables are auxiliary variables (as defined in Definition 5.7.3 below).*
- *The specification variable lists occurring in the triggered transitions are pairwise disjoint, that is* $\mathsf{s_i} \cap \mathsf{s_j} = \emptyset$ *for all* $1 \leq \mathsf{i} \neq \mathsf{j} \leq |\mathsf{tt}|$.

- *Events only deal with non controlled actions, that is*
  [Ev] $\mathsf{FAct}(\mathsf{ev_i}) \cap \mathrm{Ctr}_{\mathsf{Act}}(\mathsf{block}) = \emptyset$ *for all* $1 \leq \mathsf{i} \leq |\mathsf{tt}|$.

- *Pairs of triggered transitions are not enabled simultaneously (with regard to their assumptions) or their control sets are disjoint and one of the delays is more general:*

  [TT_Consistency] *For all* $1 \leq i \neq j \leq k$

  [TT_Consistency1] $[\![\, \mathsf{ev_i} \wedge \mathsf{ev_j} \;\Rightarrow\; \neg(\mathsf{assume_i} \wedge \mathsf{assume_j}) \,]\!]$

  or

  [TT_Consistency2a] $\mathrm{Ctr}(\mathsf{cmd_i}) \cap \mathrm{Ctr}(\mathsf{cmd_j}) = \emptyset$ *and*
  [TT_Consistency2b] $[\![\, \mathsf{ev_i} \wedge \mathsf{ev_j} \;\Rightarrow\; (\mathsf{delay_i} \Leftarrow \mathsf{delay_j}) \vee (\mathsf{delay_i} \Rightarrow \mathsf{delay_j}) \,]\!]$

- [Init1] *only controlled flexible variables occur in* init: $\mathsf{FFV}(\mathsf{init}) \subseteq \mathrm{Ctr}_V(\mathsf{block})$
  [Init2] *the initial condition is satisfiable:* $[\![\, \exists_{\mathsf{FFV(init)}}\ \mathsf{init} \,]\!]$

- *All fairness conditions are defined with regard to* Decls. *Further, for any fairness condition* $<$ fair, kind, fg, fcmd $>$

  [F1] $\mathsf{FFV}(\mathsf{fg}) \subseteq \mathrm{Ctr}_V(\mathsf{block})$
  [F2] $\mathrm{Ctr}(\mathsf{fcmd}) \subseteq \mathrm{Ctr}(\mathsf{block})$
  [F3] $[\![\, \mathsf{fg} \Rightarrow \exists_{\mathrm{Ctr}'(\mathsf{block})}\ (\ \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{\mathsf{ctr}}(\mathsf{block})\ ) \,]\!]$

  where $\delta_{\mathsf{ctr}}(\mathsf{block})$ *describes part of the next-step relation of the block as defined below in Definition 5.7.2.*

Remark 1. The control set of a block contains at least one variable, and thus $\mathrm{Ctr}(\mathsf{block}) \neq \emptyset$.

Remark 2. The control set does not contain specification variables, that is, $\mathrm{Ctr}(\mathsf{block}) \subseteq \Gamma_V \cup \Gamma_{Act}$.

Remark 3. As an immediate consequence of Definitions 5.6.6 and 5.6.7, no variable occurring primed in a transition is of class Read. Therefore, $\mathrm{class}(v) \neq \mathsf{Read}$ for all $v \in \mathrm{Ctr}_V(\mathsf{block})$ and thus, due to [Init1], also $\mathrm{class}(v) \neq \mathsf{Read}$ for all $v \in \mathsf{FV}(\mathsf{init})$.

Remark 4. Condition [TT_Consistency] required for triggered transitions may be replaced by the following weaker condition that also is sufficient, but that is harder to check:

for all $a \in \bigcup_{1 \leq j \leq |tt|} \mathrm{Ctr}(cmd_j)$ define $I_a \stackrel{\text{def}}{=} \{\ 1 \leq i \leq |tt| \mid a \in \mathrm{Ctr}(cmd_i)\ \}$;
then for all $I \subseteq \mathcal{P}(I_a)$ such that $|I| \geq 2$ one has to show:

$$[\![\bigwedge_{i \in I} ev_i \wedge g_i \Rightarrow \exists_{\bigcup_{i \in I} \mathrm{Ctr}'(cmd_i)} \bigwedge_{i \in I} \mathrm{Trl}(cmd_i)]\!].$$

Intuitively, this condition means, that any subset of transitions that may be triggered simultaneously may also be executed simultaneously.

Notation:

Block block_name

Types

T := type_definition

Parameters

param : sort(param)

Declarations

class(a) a : sort(a)

Abbreviations

Abbrev := expr

Initially init

Transitions

[gt] guard → cmd

$[\mathsf{tt}]_{\mathsf{s}}$ {assume} ⟨delay⟩ ev =▷cmd

Fairness

[wf] WF(fg,fcmd)

[sf] SF(fg,fcmd)

End

The translation of the programming notation to both logic and automata is based on a predicate describing the intended next-step relation of the block:

Definition 5.7.2 (Next-step Relation)

The next-step relation of the block block *is given by*

$$\delta(\mathsf{block}) \stackrel{def}{=} \delta_{env}(\mathsf{block}) \wedge \delta_{ctr}(\mathsf{block})$$

where

$$\delta_{env}(\mathsf{block}) \stackrel{def}{=} \bigwedge_{1 \le i \le |tt|} \forall_{s_i} \; (\; ev_i \Rightarrow g_i \wedge \mathrm{Trl}(cmd_i) \;)$$

describes the effect of the environment of the block and

$$\delta_{ctr}(\mathsf{block}) \stackrel{def}{=} \bigwedge_{a \in \mathrm{Ctr}(\mathsf{block})} (\; \mathsf{Stutter}(a) \vee \bigvee_{\{ \; i \; | \; a \in \mathrm{Ctr}(tt_i) \}} \exists_{s_i} \; ev_i \quad \vee \bigvee_{\{ \; i \; | \; a \in \mathrm{Ctr}(gt_i) \}} guard_i \wedge \mathrm{Trl}(cmd_i) \;)$$

describes the effect of the behavior controlled by the block itself.

History variables are intended to be used to record the occurrence of events. Thus, their value should be determined by the history of past events. This is achieved in Definition 5.7.1 by forcing all history variables of a block to be auxiliary variables. Auxiliary variables are defined semantically by means of two functions determining a unique initial value and a unique "primed" value based on the actual value of the variable and the values of the visible actions.

Definition 5.7.3 (Auxiliary Variables)

A variable h *is called auxiliary* **iff** *there are two functions* f^0 *and* f *such that for all* $(\xi_0, \alpha_0, \xi_1, \alpha_1, \xi_2, \dots) \in$ **Traces**$(\mathsf{init} \wedge \Box\delta(\mathsf{block}))$ *it holds that*

$$\xi_0(h) \;=\; f^0() \;\;,$$
$$\xi_{i+1}(h) \;=\; f(\xi_i(h)\;,\; \langle\; \alpha_i(A) \mid A \in \mathsf{vActs} \;\rangle\;) \text{ and}$$
$$\xi_{i+1}(h) \;=\; \xi_i(h) \text{ whenever } \alpha_i(A) = \bot \text{ for all actions } A \in \mathsf{vActs}.$$

One frequently used special kind of block is worth mentioning:

Definition 5.7.4 (Interfaces)
Blocks that consist of only triggered transitions and such that $\mathrm{class}(\mathrm{Ctr}(\mathsf{block})) \subseteq \{\mathsf{History}\}$ *are called interfaces. An interface* I *can be* inverted, *denoted by* $\bar{I}$, *by inverting the environment class of all actions occurring in the interface (*Out*-actions become* In*-actions and vice versa).*

Corollary 5.7.5
Traces$(I) =$ **Traces**$(\bar{I})$ *for any interface* I

Two lemmas, stating that blocks are stutter invariant and input enabled, conclude this section:

The first lemma states that the transition relation $\delta(\mathsf{block})$ may be satisfied by stuttering. The proof is purely technical and moved to the appendix.

Lemma 5.7.6 (Stutter Invariance)

Let ξ *be an arbitrary valuation of* Vars, *and* ξ' *a valuation of* Vars' *such that* $\xi'(x') = \xi(x)$ *for all* $x \in \mathsf{Vars}$.

Then $[\![\, \delta(\mathsf{block}) \,]\!]^{(\xi,\tau,\xi')}$ *or, equivalently,* $[\![\, (\bigwedge_{a \in \mathsf{Vars} \cup \mathsf{Acts}} \mathsf{Stutter}(a)\;) \;\Rightarrow\; \delta(\mathsf{block}) \,]\!]$.

The second lemma, whose proof also is postponed, states that the transition relation $\delta(\mathsf{block})$ may be satisfied regardless of the values of non-controlled actions and variables (given events only occur in states allowed by their corresponding guards).

Lemma 5.7.7 (Input Enabledness)

$$[\![\, (\bigwedge_{1 \le i \le |tt|} \forall_{s_i}(\; ev_i \Rightarrow g_i \;)\;) \;\Rightarrow\; \exists_{\mathrm{Ctr}'(\mathsf{block})}\delta(\mathsf{block}) \,]\!]$$

5.7.1 Translating Blocks

In this section, the textual representation for blocks presented in the previous section gets translated into a temporal logic formula and into a (concrete) BTS. The main theorem then states that both of these "models" yield the same set of traces.

Definition 5.7.8 (Translation to Logic)
The temporal formula representing the "safety" part:

$$\Delta(\mathsf{block}) \stackrel{def}{=} \mathsf{init} \wedge \Box\delta(\mathsf{block})$$

The "full" temporal formula:

$$\Phi(\mathsf{block}) \stackrel{def}{=} \Delta(\mathsf{block}) \wedge \bigwedge_{<\ \mathsf{fair,\ WF,\ fg,\ fcmd}\ >\in \mathsf{Fair}} \mathcal{WF}(\mathsf{fg}, \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd})) \wedge \bigwedge_{<\ \mathsf{fair,\ SF,\ fg,\ fcmd}\ >\in \mathsf{Fair}} \mathcal{SF}(\mathsf{fg}, \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd})) \wedge \mathcal{WF}\Big(\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|} \mathsf{guard_i}, \bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|} (\ \mathsf{guard_i} \wedge \mathrm{Trl}(\mathsf{cmd_i})\ )\ )$$

The last conjunct is referred to as local progress.

The reason for distinguishing between $\Delta(\mathsf{block})$ and $\Phi(\mathsf{block})$ is justified by the fact that the verification of safety properties can be done completely without referring to the fairness conditions: Suppose some safety property $\Box\psi$ holds for all (fair) traces satisfying $\Phi(\mathsf{block})$, but not for all traces satisfying $\Delta(\mathsf{block})$. Then there is a trace in $\mathbf{Traces}(\Delta(\mathsf{block}))$ and a position i on that trace such that $[\![\ \neg\psi\ ]\!]^{(\xi_i,\alpha_i,\xi_{i+1})}$. But all initial segments of traces can be completed to fair traces[3]. Thus

Corollary 5.7.9

For any transition predicate ψ,

$$\models \Delta(\mathsf{block}) \Rightarrow \Box\psi \quad \textbf{iff} \quad \models \Phi(\mathsf{block}) \Rightarrow \Box\psi$$

or, equivalently,

$$\mathbf{Traces}(\Delta(\mathsf{block})) \subseteq \mathbf{Traces}(\Box\psi) \quad \textbf{iff} \quad \mathbf{Traces}(\Phi(\mathsf{block})) \subseteq \mathbf{Traces}(\Box\psi) \ .$$

The following lemma shows that local progress might also be stated directly on the textual level as an explicit fairness conditions satisfying the conditions [F1], [F2] and [F3].

Lemma 5.7.10

Local progress may also be expressed explicitly as the fairness conditions

$$<\ \mathsf{LP,\ WF},\ \bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|} \mathsf{guard_i},\ \bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|} \mathsf{guard_i} \wedge \mathrm{Trl}(\mathsf{cmd_i})\ >.$$

The proof of this lemma is given in the appendix.

In the sequel, let $\mathsf{Fair}^{\mathsf{LP}}$ denote the set

$$\mathsf{Fair} \cup \{<\ \mathsf{LP,\ WF},\ \bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|} \mathsf{guard_i},\ \bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|} \mathsf{guard_i} \wedge \mathrm{Trl}(\mathsf{cmd_i})\ >\} \ .$$

Next, a translation to a standard BTS $\Sigma(\mathsf{block})$ is given:

[3] This was shown in Chapter 4 for the traces of BTS, but the same scheduler also works for the temporal formula $\Phi(\mathsf{block})$. Actually, it will soon be shown that $\mathbf{Traces}(\Phi(\mathsf{block}))$= $\mathbf{Traces}(\Sigma(\mathsf{block}))$ for a suitable BTS $\Sigma(\mathsf{block})$.

Definition 5.7.11 (Translation to BTS)

$\Sigma(\mathsf{block}) \stackrel{def}{=} (V, I, \mathcal{B}, st, M, \mathcal{F})$

where

$$V = \prod_{x \in \mathrm{Ctr}_V(\mathsf{block})} \mathcal{D}(x) \quad ,$$

$$I = \{\zeta \in V \mid [\![\, \mathsf{init} \,]\!]^{\zeta}_{\mathcal{I},\gamma} \ \} \quad ,$$

$$\mathcal{L} = \prod_{x \in \mathsf{Vars}} \mathcal{D}(x) \times \prod_{A \in \mathsf{Acts}} \mathcal{D}_{\perp}(A) \times \prod_{x' \in \mathsf{Vars}'} \mathcal{D}(x') \quad ,$$

$$\mathcal{B} = \mathcal{P}(\mathcal{L}) \quad ,$$

$$st = \{(\chi, \tau, \chi') \in \mathcal{L} \mid \chi(x) = \chi'(x') \textit{ for all } x \in \mathsf{Vars}\} \quad ,$$

$$M(v,w) = \{\eta \in \mathcal{L} \mid \eta(x) = v(x) \textit{ for all } x \in \mathrm{Ctr}_V(\mathsf{block}), \ \eta(x') = w(x) \textit{ for all } x' \in \mathrm{Ctr}'_V(\mathsf{block}) \textit{ , and } [\![\, \delta(\mathsf{block}) \,]\!]^{\eta} \ \}$$

$$\mathcal{F} = \{ \ \mathsf{F}(E, L) \mid < \mathsf{fair},\ \mathsf{F},\ \mathsf{fg},\ \mathsf{fcmd} > \in \mathsf{Fair}^{\mathsf{LP}} \ ,$$

$$E = \{(v,w) \in V \times V \mid \textit{exists } \mu \textit{ such that } \begin{cases} \mu(x) = v(x) & , x \in \mathrm{Ctr}_V(\mathsf{block}) \\ \mu(x') = w(x) & , x' \in \mathrm{Ctr}'_V(\mathsf{block}) \end{cases} \textit{and } [\![\, \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block}) \,]\!]^{\mu} \ \} \textit{ , and}$$

$$L(v,w) = \{\eta \in \mathcal{L} \mid [\![\, \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block}) \,]\!]^{\eta} \textit{ and } \begin{cases} \eta(x) = v(x) & , x \in \mathrm{Ctr}_V(\mathsf{block}) \\ \eta(x') = w(x) & , x' \in \mathrm{Ctr}'_V(\mathsf{block}) \end{cases} \ \} \textit{ for all } (v,w) \in E \ \}$$

Lemma 5.7.12 $\Sigma(\mathsf{block})$ *is well-defined.*

Proof

1. $I \subseteq V$ by definition.

2. $\emptyset \neq I$:
   By [Init2], $[\![\, \exists_{\mathsf{FFV(init)}}\, \mathsf{init} \,]\!]$, that is, there is some valuation β of the variables in $\mathsf{FFV(init)}$ such that $[\![\, \mathsf{init} \,]\!]^{\beta}$. Thus, $[\![\, \mathsf{init} \,]\!]^{\xi}$ for any $\xi : \mathrm{Ctr}_V(\mathsf{block}) \to \mathcal{U}$ that agrees with β on $\mathsf{FFV(init)} \stackrel{[\mathrm{Init1}]}{\subseteq} \mathrm{Ctr}_V(\mathsf{block})$.

3. $\mathcal{B}$ is a complete, atomic Boolean algebra by definition.

4. $M : V \times V \to \mathcal{B}$ by definition.

5. $st \cap M(v,v) \neq \emptyset$ for all $v \in V$: Due to lemma 5.7.6, it is sufficient to define $\mu(x) \stackrel{\text{def}}{=} \begin{cases} v(x) & \text{, for } x \in \mathrm{Ctr}_V(\mathsf{block}) \\ \text{arb.} & \text{, for } x \notin \mathrm{Ctr}_V(\mathsf{block}) \end{cases}$. Then $(\mu, \tau, \mu) \in st \cap M(v,v)$.

6. $\emptyset \neq L(e) \cap M(e)$ for all $e = (v,w) \in E$:
   If $e \in E$ then there exists some μ such that $[\![\mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block})]\!]^{\mu}$ and $\mu(x) = \begin{cases} v(x) & , x \in \mathrm{Ctr}_V(\mathsf{block}) \\ w(x) & , x \in \mathrm{Ctr}'_V(\mathsf{block}) \end{cases}$. Then define $\nu(a) = \begin{cases} \bot & , a \in \mathsf{Acts} - \mathrm{Ctr}(\mathsf{block}) \\ \mu(a) & \text{, else} \end{cases}$.

   The actions from $\mathsf{Acts} - \mathrm{Ctr}(\mathsf{block})$ do not occur in $\mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block})$. Therefore, $[\![\mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block})]\!]^{\nu}$.
   On the other side, besides $[\![\delta_{ctr}(\mathsf{block})]\!]^{\nu}$ also $[\![\delta_{env}(\mathsf{block})]\!]^{\nu}$ holds due to Corollary 5.6.3, because all actions occurring in events are elements from $\mathsf{Acts} - \mathrm{Ctr}(\mathsf{block})$ (which is due to condition [Ev] in Definition 5.7.1). That is, $\nu \in M(e)$.

■

Due to Theorem 4.2.11,

Corollary 5.7.13 $\mathbf{Traces}(\Sigma(\mathsf{block})) \neq \emptyset$.

Furthermore, both the translation to logic and the translation to BTS result in the same set of traces as shown in Figure 5.1 and expressed in the following theorem:

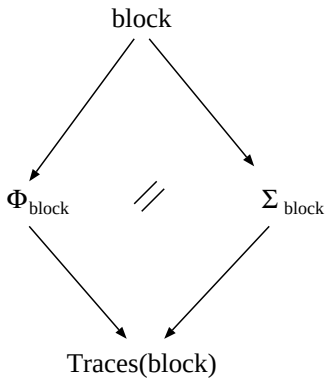


Figure 5.1: The two models of block yield the same observable behavior.

Theorem 5.7.14 (Equivalence of models)

$\mathbf{Traces}(\mathsf{block}) \stackrel{def}{=} \mathbf{Traces}(\Phi(\mathsf{block})) = \mathbf{Traces}(\Sigma(\mathsf{block}))$

Proof:
Let $\sigma = \xi_0 \xrightarrow{\alpha_0} \xi_1 \xrightarrow{\alpha_1} \xi_2 \cdots$. Then

$\sigma \in \mathbf{Traces}(\Phi(\mathsf{block}))$

iff $\sigma \models \Phi(\mathsf{block})$

iff

[1] $[\![\,\mathsf{init}\,]\!]^{\xi_0}$

[2] $[\![\,\delta(\mathsf{block})\,]\!]^{(\xi_i,\alpha_i,\xi_{i+1})}$ for all i

[3a] infinitely often $[\![\,\neg\mathsf{fg}\,]\!]^{\xi_i}$ or infinitely often $[\![\,\mathsf{fg}\wedge\mathrm{Trl}(\mathsf{cmd})\,]\!]^{(\xi_i,\alpha_i,\xi_{i+1})}$
for all $<$ fair, WF, fg, fcmd $>\in \mathsf{Fair}^{\mathsf{LP}}$

[3b] infinitely often $[\![\,\mathsf{fg}\,]\!]^{\xi_i}$ implies infinitely often $[\![\,\mathsf{fg}\wedge\mathrm{Trl}(\mathsf{cmd})\,]\!]^{(\xi_i,\alpha_i,\xi_{i+1})}$
for all $<$ fair, SF, fg, fcmd $>\in \mathsf{Fair}^{\mathsf{LP}}$

On the other hand,

$\sigma \in \mathbf{Traces}(\Sigma(\mathsf{block}))$
iff there is a sequence $(v_0, v_1, \dots) \in V^\omega$ such that

[I] $v_0 \in I$

[II] $(\xi_i, \alpha_i, \xi_{i+1}) \in M(v_i, v_{i+1})$ for all i

[IIIa] infinitely often $v_i \notin G$ or infinitely often $(v_i, v_{i+1}) \in E$ and $(\xi_i, \alpha_i, \xi_{i+1}) \in L(v_i, v_{i+1})$ for all $\mathsf{WF}(G, E, L) \in \mathcal{F}$

[IIIb] infinitely often $v_i \in G$ implies infinitely often $(v_i, v_{i+1}) \in E$ and $(\xi_i, \alpha_i, \xi_{i+1}) \in L(v_i, v_{i+1})$ for all $\mathsf{SF}(G, E, L) \in \mathcal{F}$

To show [1] **iff** [I], [2] **iff** [II], [3a] **iff** [IIIa] and [3b] **iff** [IIIb], we define $v_i \stackrel{\text{def}}{=} \xi_i|_{\mathrm{Ctr}_V(\mathsf{block})}$.

[1]
iff $[\![\,\mathsf{init}\,]\!]^{\xi_0}$
iff /* $\mathsf{FFV}(\mathsf{init}) \subseteq \mathrm{Ctr}(\mathsf{block})$ */
$[\![\,\mathsf{init}\,]\!]^{v_0}$
iff $v_0 \in I$
iff [I]

[2]
iff $[\![\,\delta(\mathsf{block})\,]\!]^{(\xi_i,\alpha_i,\xi_{i+1})}$ for all i
iff /* $v_i(x) \stackrel{\text{def}}{=} \xi_i(x)$ for all $x \in \mathrm{Ctr}_V(\mathsf{block})$ */
$(\xi_i, \alpha_i, \xi_{i+1}) \in M(v_i, v_{i+1})$ for all i
iff [II]

Instead of proving [3a] **iff** [IIIa] and [3b] **iff** [IIIb] directly, we show (stronger) that

$[\![\,\mathsf{fg}\,]\!]^{\xi_i}$ **iff** $v_i \in G$

and

$[\![\,\mathsf{fg}\wedge\mathrm{Trl}(\mathsf{fcmd})\,]\!]^{(\xi_i,\alpha_i,\xi_{i+1})}$ **iff** $(v_i, v_{i+1}) \in E$ and $(\xi_i, \alpha_i, \xi_{i+1}) \in L(v_i, v_{i+1})$.

$[\![\,\mathsf{fg}\,]\!]^{\xi_i}$
iff /* $\mathsf{FFV}(\mathsf{fg}) \subseteq \mathrm{Ctr}(\mathsf{block})$ */
$[\![\,\mathsf{fg}\,]\!]^{v_i}$
iff /* [F3] */
there are v, μ such that $\mu|_{\mathrm{Ctr}_V(\mathsf{block}) \times \mathrm{Ctr}_{V'}(\mathsf{block})} = (v_i, v)$ and $[\![\,\mathsf{fg}\wedge\mathrm{Trl}(\mathsf{fcmd})\wedge\delta_{\mathsf{ctr}}(\mathsf{block})\,]\!]^{\mu}$

iff $v_i \in G$

$[\![\mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd})]\!]^{(\xi_i,\alpha_i,\xi_{i+1})}$

iff $/* \; [\![\delta(\mathsf{block})]\!]^{(\xi_i,\alpha_i,\xi_{i+1})}$ for all i $*/$

$[\![\mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{\mathsf{ctr}}(\mathsf{block})]\!]^{(\xi_i,\alpha_i,\xi_{i+1})}$

iff $(v_i, v_{i+1}) \in E$ and $(\xi_i, \alpha_i, \xi_{i+1}) \in L(v_i, v_{i+1})$

■

5.7.2 Some Remarks Concerning Fairness

Fairness is the most demanding concept in the theory of TLT. Nissim Francez wrote a whole book ([Fra86]) that is dedicated solely to fairness. The following remarks aim at providing at least some deeper insight into the fairness notion of TLT and the consistency conditions required for fairness conditions.

REMARK 1. Local progress is not the "weakest possible fairness". See Figure 5.2. Local progress is used because of its very intuitive meaning, namely, "if continuously some guarded transition is enabled then eventually (at least) one enabled guarded transition gets executed (scheduled)". Thus, local progress states that the guarded transitions of a block may not starve. In most examples this is the sole fairness condition necessary.

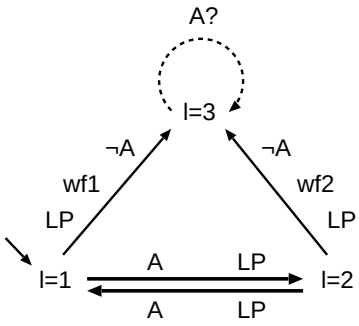
Block Local_Progress

Declarations

In	A	: ()	
Local	l	: { 1, 2, 3 }	init 1

Transitions

‖	A	=▷	If l≠3 Then l′=3-l
‖	l=1	→	l′=3
‖	l=2	→	l′=3



Fairness

[wf1]	WF(l=1 ,l′=3)
[wf2]	WF(l=2 ,l′=3)

End

Figure 5.2: $\Box\Diamond \mathsf{l} = 3$ holds only due to local progress. In fact, the two weak fairness conditions do not restrict the set of traces at all (whereas local progress does) and might as well be omitted. In the terminology of Francez, wf1 and wf2 are not liveness enhancing.

REMARK 2. The necessity of the condition [F1] may be seen considering the block shown in Figure 5.3.
In this example, the fairness constraint intuitively causes a Reset signal to be sent out if r continues to be 1. That is, in all traces infinitely often $\mathsf{r} \neq 1$ or Reset occurs infinitely often.

```
Block  Flout_F1
  Declarations
    Out      Reset   : ()
    Read     r       : Bit
  Transitions
    ‖   r=1   → Reset
  Fairness
    [wf1]WF(r=1 ,Reset)
End
```

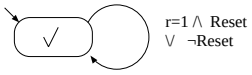


Figure 5.3: An example motivating [F1].

This is captured in the logical view by $\Phi(\mathsf{Flout\_F1})$. The corresponding BTS $\Sigma(\mathsf{Flout\_F1})$ however further forbids ,e.g., the trace

$$r = 0 \xrightarrow{Reset=\bot} 0 \xrightarrow{\bot} 0 \xrightarrow{\bot} 0 \xrightarrow{\bot} 0 \xrightarrow{\bot} \ldots$$

because $G = V$, forcing an edge labeled with $\mathsf{r} = 1 \wedge \mathsf{Reset}$ to occur infinitely often. Actually, this is due to an "inaccurate" fairness definition for BTS. To be exact, a fairness constraint $\mathsf{F(E, L)}$ defines two implicit "guards" $\mathsf{G_E}$ and $\mathsf{G_L}$ being the projections of E to its first component (the old G) and the projection of L to the unprimed variables of the labeling universe: $\mathsf{G_L(e)} \stackrel{\text{def}}{=} \{\ \xi \mid \text{ exists } \alpha, \xi' \text{ such that } [\![\, \mathsf{L(e)} \,]\!]^{(\xi,\alpha,\xi')}\}$. The fairness then is defined to be enabled **iff** it is enabled with respect to both $\mathsf{G_E}$ and $\mathsf{G_L}$. However, this can not be generalized in a straightforward way to arbitrary labeling algebras. Further, the example seems to be rather artificial and can be circumvented by holding a local copy r_local of the Read variable r as shown in Figure 5.4.

REMARK 3. [F3] forbids obviously senseless fairness conditions like $\mathsf{WF(true, x' = 0 \wedge x' = 1)}$ that trivially would result in non executable specifications. Furthermore, [F3] forbids fairness conditions that contradict the transition relation $\delta(\mathsf{block})$:

```
Block  Contradiction
  Declarations
    Write      x      : Natural  init 0
  Transitions
    ‖   true  →  x′=x+1
  Fairness
    ‖   WF(true,x′=x+2)
End
```

Block Obey_F1

Declarations

Out	Reset	: ()
Read	r	: Bit
Local	r_local	: Bit

Transitions

[] true → r_local′=r

[] r_local=1 → Reset

Fairness

[wf1]WF(r_local=1,Reset)

End

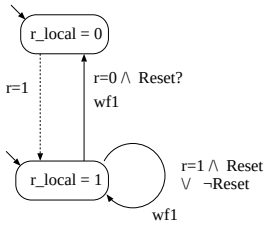


Figure 5.4: Block Obey_F1 is well-defined. Furthermore, with regard to the visible variables and actions, Obey_F1 has exactly the desired traces, that is, $\mathbf{Traces}_{\{r\},\{\mathsf{Reset}\}}(\mathsf{Obey\_F1}) = \mathbf{Traces}(\Phi(\mathsf{Flout\_F1}))$.

The transition predicate $\delta_{ctr}(\mathsf{Contradiction})$ is given by $x \neq x' \Rightarrow x' = x + 1$. The fairness condition "always eventually increase x by 2" contradicts the fact, that x may only be incremented by one. This is reflected by condition [F3] which requires $[\![\exists_{x'} (x' = x + 2 \ \wedge \ \delta_{\mathsf{ctr}}(\mathsf{Contradiction}))]\!]$ to hold. However, $\exists_{x'} (x' = x + 2 \ \wedge \ (x \neq x' \vee x' = x + 1))$ is not even satisfiable.

Remark 4. The expressiveness of the fairness notion as advocated in this thesis is demonstrated by the following example:

Block Fair

Declarations

Out	A	: { 1,2 }
Write	x	: Natural init 0

Transitions

[] true → x′=x+1 || A(1)

[] true → x′=x+1 || A(2)

Fairness

[] WF(true,A(1))

End

This results in the BTS shown in Figure 5.5.
Clearly, $(0 \xrightarrow{2} 1 \xrightarrow{2} 2 \xrightarrow{2} 3 \xrightarrow{2} 4 \cdots) \notin \mathbf{Traces}(\Phi(\mathsf{Fair}))$. If the notion of fairness was based on runs only (as for example in [Kur94]), such specialized conditions could not be expressed.

Figure 5.5: BTS corresponding to block Fair. All transitions are colored by both local progress and WF(true,A(1)).

5.7.3 Abstract Models

The abstract models of a block are characterized by their states being state predicates forming an arbitrary partition of the concrete state space.

Definition 5.7.15 (Translation to Abstract BTS)
Let W be a set of state predicates p such that ,

- [aBTS1] $\mathsf{FFV}(p) \in \mathrm{Ctr}_V(\mathsf{block})$ *and* $[\![\exists_{\mathrm{Ctr}_V(\mathsf{block})}\ p]\!]$
- [aBTS2] $[\![\neg(p \wedge q)]\!]$ *for arbitrary* $p, q \in W$ *with* $p \neq q$
- [aBTS3] $[\![\bigvee_{p \in W} p]\!]$

Then the abstract BTS $\Sigma^a(\mathsf{block}) \stackrel{def}{=} (W, J, \mathcal{B}, st, N, \mathcal{G})$ *of a block* block *based on* W *is given by*

$$J = \{ w \in W \mid [\![\exists_{\mathrm{Ctr}_V(\mathsf{block})}(w \wedge \mathsf{init})]\!] \} \ ,$$

$$\mathcal{L} = \prod_{x \in \mathsf{Vars}} \mathcal{D}(x) \times \prod_{A \in \mathsf{Acts}} \mathcal{D}_\perp(A) \times \prod_{x' \in \mathsf{Vars}'} \mathcal{D}(x') \ ,$$

$$\mathcal{B} = \mathcal{P}(\mathcal{L}) \ ,$$

$$st = \{ (\chi, \tau, \chi') \in \mathcal{L} \mid \chi(x) = \chi'(x') \text{ for all } x \in \mathsf{Vars} \} \ ,$$

$$N(p,q) = \{ \eta \in \mathcal{L} \mid [\![\delta(\mathsf{block}) \wedge p \wedge q']\!]^\eta \}$$

$$\begin{aligned} \mathcal{G} = \{\ \mathsf{F}(E,L) \mid &< \mathsf{fair}, \mathsf{F}, \mathsf{fg}, \mathsf{fcmd} > \in \mathsf{Fair}^{LP} \ , \\ &E = \{ (p,q) \in W \times W \mid [\![p \Rightarrow \mathsf{fg}]\!] \text{ and} \\ &\qquad \text{exists } \mu \text{ such that } [\![p \wedge q' \wedge \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block})]\!]^\mu \} \ , \\ &L(p,q) = \{ \eta \in \mathcal{L} \mid [\![p \wedge q' \wedge \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block})]\!]^\eta \} \quad \text{for all } (p,q) \in E \quad \} \end{aligned}$$

Lemma 5.7.16 $\Sigma^a(\mathsf{block})$ *is well-defined.*

The proof of this lemma is similar to the proof of Lemma 5.7.12 and is given in the appendix.

■

Theorem 5.7.17 (Relationship between models)

$\mathbf{Traces}(\Sigma(\mathsf{block})) \subseteq \mathbf{Traces}(\Sigma^a(\mathsf{block}))$

Proof:
This proof can be done either directly or by use of Theorem 4.2.26. A direct proof is given in the appendix. Here, it is shown that the premises of Theorem 4.2.26 are fulfilled. Transcribed to the situation above this means to find a function Φ such that

(1) Φ maps $v \in V$ to $w \in W$

(2) Φ maps $v_0 \in I$ to $w_0 \in J$

(3) Φ maps atoms $\eta \in \mathcal{L}$ to atoms $\eta \in \mathcal{L}$ and arbitrary elements $b \in \mathcal{B}$ to $\Phi(b) = \bigcup_{\eta \in b} \Phi(\eta)$,

(4) $N(w_1, w_2) \supseteq \bigcup_{\{(v_1,v_2) \in V \times V \mid \Phi(v_1)=w_1 \text{ and } \Phi(v_2)=w_2\}} \Phi(M(v_1, v_2))$

(5) for any acceptance condition $\mathsf{F}(E^a, L^a) \in \mathcal{G}$ there is an acceptance condition $\mathsf{F}(E, L) \in \mathcal{F}$ such that

$$E^a \subseteq \{ (w_1, w_2) \in W \times W \mid \text{exists } (v_1, v_2) \in E \text{ such that } \Phi(v_1) = w_1 \text{ and } \Phi(v_2) = w_2 \}$$

and

$$L^a(w_1, w_2) \supseteq \bigcup_{\{(v_1,v_2) \in E \mid \Phi(v_1)=w_1 \text{ and } \Phi(v_2)=w_2\}} \Phi(L(v_1, v_2)).$$

Firstly, let $\Phi(v)$ be the (uniquely determined) $w \in W$ such that $[\![w]\!]^v$.

Secondly, if $v_0 \in I$, then by definition $[\![\mathsf{init}]\!]^{v_0}$. Then $[\![w_0 \wedge \mathsf{init}]\!]^{v_0}$. Thus, $[\![\exists_{\mathrm{Ctr}_V(\mathsf{block})} (w_0 \wedge \mathsf{init})]\!]$, i.e. $w_0 \in J$.

Choosing the identity function for Φ, that is, by defining $\Phi(\eta) \stackrel{\text{def}}{=} \eta$, the extra condition of the third item follows from Lemma 4.1.10.

Let $\eta \in M(v_1, v_2)$. Then, $[\![\delta(\mathsf{block})]\!]^\eta$, $v_1(x) = \eta(x)$ and $v_2(x) = \eta(x')$ for all $x \in \mathrm{Ctr}_V(\mathsf{block})$. Thus $[\![\delta(\mathsf{block}) \wedge w_1 \wedge {w_2}']\!]^\eta$ by definition of Φ, i.e. $\eta \in N(w_1, w_2)$. Thus (4).

Suppose $(w_1, w_2) \in E^a$. That is, $[\![w_1 \Rightarrow \mathsf{fg}]\!]$ and there is some μ such that $[\![w_1 \wedge {w_2}' \wedge \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block})]\!]^\mu$. Defining $v_1(x) \stackrel{\text{def}}{=} \mu(x)$ and $v_2(x) \stackrel{\text{def}}{=} \mu(x')$ for all $x \in \mathrm{Ctr}_V(\mathsf{block})$, it follows that $(v_1, v_2) \in E$ and

$(w_1, w_2) \subseteq \{ (w_1, w_2) \in W \times W \mid \text{exists } (v_1, v_2) \in E \text{ such that } \Phi(v_1) = w_1 \text{ and } \Phi(v_2) = w_2 \}$.

It remains to show that for any pair $(v_1, v_2) \in E$ such that $[\![w_1]\!]^{v_1}$ and $[\![w_2]\!]^{v_2}$ it holds that $\eta \in L(v_1, v_2)$ implies $\eta \in L^a(w_1, w_2)$:
If $\eta \in L(v_1, v_2)$ then $[\![\mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block})]\!]^\eta$, $v_1(x) = \eta(x)$ and $v_2(x) = \eta(x')$ for all $x \in \mathrm{Ctr}_V(\mathsf{block})$. Since $[\![w_1]\!]^{v_1}$ and $[\![w_2]\!]^{v_2}$ it therefore follows that $\eta \in L^a(w_1, w_2)$.

Thus, by Theorem 4.2.26, it holds that $\Phi(\mathbf{Traces}(\Sigma(\mathsf{block}))) \subseteq \mathbf{Traces}(\Sigma^a(\mathsf{block}))$ and because $\Phi = id$, $\mathbf{Traces}(\Sigma(\mathsf{block})) \subseteq \mathbf{Traces}(\Sigma^a(\mathsf{block}))$. ■

The simple counterexample presented in Figure 5.6 shows that the opposite is not true in general.

In the theorem stated above, the same Boolean algebra was chosen for the labeling of the concrete BTS and of the abstract BTS. This simplifies the proof, because the identity function can be used as Φ. In general, however, it will be necessary not only to reduce the set of states but also the set of labels. As already mentioned in the remarks following Theorem 4.2.26, the canonical projections form natural candidates. Indeed, from Lemma 4.2.24,

Corollary 5.7.18

Let $\mathsf{aVars} \subseteq \mathsf{Vars}$, $\mathsf{aActs} \subseteq \mathsf{Acts}$ *and* $\mathcal{L}^a = \prod_{x \in \mathsf{aVars}} \mathcal{D}(x) \times \prod_{A \in \mathsf{aActs}} \mathcal{D}_\perp(A) \times \prod_{x \in \mathsf{aVars}} \mathcal{D}(x)$.

Then $\quad \mathbf{Traces}(\Sigma^a(\mathsf{block})\big|_{\mathcal{L}^a}) = \mathbf{Traces}(\Sigma^a(\mathsf{block}))\big|_{\mathcal{L}^a}$

and thus $\quad \mathbf{Traces}(\Sigma(\mathsf{block}))\big|_{\mathcal{L}^a} \subseteq \mathbf{Traces}(\Sigma^a(\mathsf{block})\big|_{\mathcal{L}^a})$.

Block StrangeTraces
Declarations
Write x : Natural Init 0
Transitions
[] true → x′=x+1
End

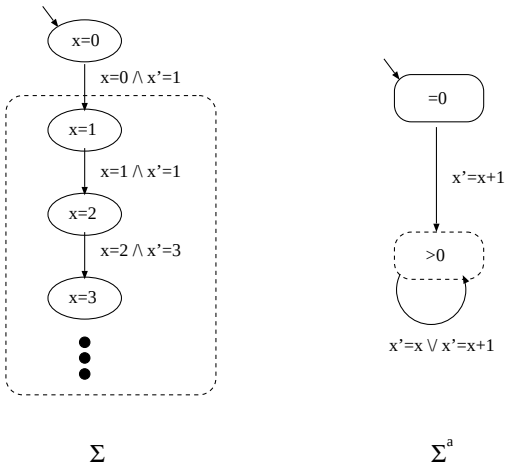


Figure 5.6: The abstract BTS shown on the right consists of only two states, one allows 0 as sole valuation whereas the second one represents all natural numbers greater than 0.
Let $\tau = ((0,,1),(7,,8),(2,,3),(4,,4),\dots)$. Then $\tau \in \mathbf{Traces}(\Sigma^a(\mathsf{block})) - \mathbf{Traces}(\Sigma(\mathsf{block}))$.

The question that immediately arises in the context of abstract models is how to find "good" ones. Typically this is a creative task, whereas checking whether a given abstract model is well defined often is significantly easier. In TLT, it is possible to extract control information directly from the textual specification. This information is used to define the necessary partition of the state space. Even though such an automated construction may fail to result in models that can be used to verify the desired properties, it often serves as a good starting point. The resulting models may be further refined by splitting states either automatically (calculating for example weakest preconditions) or by hand. Often, however, it is easier to adjust the specifications by splitting transitions and thus introducing additional guards or by replacing actions by signals (both transformations typically add control information). As a side effect, the control flow is then also more "obvious", that is the textual specifications profit by getting more readable.

Definition 5.7.19 (Control Predicates and Symbolic Guards)
The set of control predicates of a block is defined as:

$$\mathsf{CP} \stackrel{def}{=} \{\mathsf{init}\} \cup \{\, \mathsf{g_i} \mid 1 \le i \le l\} \cup \{\mathsf{fg} \mid < \mathsf{fair}, \mathsf{F}, \mathsf{fg}, \mathsf{fcmd} > \in \mathsf{Fair}\}$$

The set of literals based upon CP *is defined as:* $\quad \mathsf{LIT(CP)} \stackrel{def}{=} \{ \bigwedge_{p \in K} p \wedge \bigwedge_{p \in L} \neg p \mid K \dot\cup L = \mathsf{CP}\}$

The set of symbolic guards is then defined as: $\mathsf{symbGuards} \stackrel{def}{=} \{\ l \in \mathsf{LIT(CP)} \mid [\![\exists_{\mathsf{FFV}(l)}\ l]\!]\ \}$

That is, all state predicates occurring in the textual representation are used as control predicates. Then, all possible conjunctions are formed in which all these predicates occur either negated or unnegated. As there are only finitely many control predicates, this set is also finite (with $2^{|\mathsf{CP}|}$ elements). Finally, the set of symbolic guards is obtained by removing all conjunctions that are not satisfiable.

Lemma 5.7.20
symbGuards *defines a partition of the state space:*

1. $\mathsf{FFV}(G) \subseteq \mathrm{Ctr}_V(\mathsf{block})$ *for all* $G \in \mathsf{symbGuards}$
2. $[\![\neg(G_1 \wedge G_2)]\!]$ *for arbitrary* $G_1, G_2 \in \mathsf{symbGuards}$ *with* $G_1 \neq G_2$
3. $[\![\bigvee_{G \in \mathsf{symbGuards}} G]\!]$

Proof:

1. By definition of the control predicates.
2. Suppose there is some valuation ξ such that $[\![\neg(G_1 \wedge G_2)]\!]^\xi$ for some $G_1 \neq G_2$. Then, there has to be some $p \in \mathsf{CP}$ such that p occurs positive in G_1 but negative in G_2. Then $[\![p]\!]^\xi$ and $[\![\neg p]\!]^\xi$. Contradiction.
3. Let ξ be an arbitrary valuation. Then for any $p \in \mathsf{CP}$ either $[\![p]\!]^\xi$ or $[\![\neg p]\!]^\xi$. Let $K \stackrel{\text{def}}{=} \{p \in \mathsf{CP} \mid [\![p]\!]^\xi\}$ and $L \stackrel{\text{def}}{=} \{p \in \mathsf{CP} \mid [\![\neg p]\!]^\xi\}$. Then $[\![\bigwedge_{p \in K} p \wedge \bigwedge_{p \in L} \neg p]\!]^\xi$ and $\bigwedge_{p \in K} p \wedge \bigwedge_{p \in L} \neg p \in \mathsf{symbGuards}$. Especially, $[\![\bigvee_{G \in \mathsf{symbGuards}} G]\!]^\xi$.

■

Definition 5.7.21 (Symbolic BTS)
Using the notation of definition 5.7.15 and 5.7.19,

$$\Sigma^s(\mathsf{block}) \stackrel{def}{=} (\mathsf{symbGuards}, J, \mathcal{B}, st, N, \mathcal{G})$$

is called the symbolic BTS corresponding to block.

Corollary 5.7.22 (Relationship between models)

1. $\Sigma^s(\mathsf{block})$ *is an abstract BTS of* block
2. $J = \{G \in \mathsf{symbGuards} \mid \mathsf{init}$ *occurs positive in* $G\ \}$
3. $\mathbf{Traces}(\Sigma(\mathsf{block})) \subseteq \mathbf{Traces}(\Sigma^s(\mathsf{block}))$

In the worst case, there are 2^n symbolic guards for n control predicates. As an example consider the control predicates $x = 0$, $y = 0$ and $z = 0$. However, typically many of the control predicates contradict each other. For example $x = 1$, $x = 2$ and $x = 3$ exclude each other pairwise and result in the 4 symbolic guards $x = 1$, $x = 2$, $x = 3$ and $x \neq 1 \wedge x \neq 2 \wedge x \neq 3$.

To analyze the complexity in case of contradictory control predicates, consider the following algorithm to compute symbolic Guards:

Block Symbolic_Guards

Parameters

n	: Natural	% number of control predicates
ctr_pred	: Array(1,n) OF FOL_Predicates	% n control predicates

Declarations

Local sel	: Array(1,n) Of {pos, neg}	% sel(i)=pos **iff** % the i-th ctr_pred enters the symb. % guard currently tested positiv
Local d	: Natural	% length (depth) of symb. guard % currently tested
Write symGuardL	: List Of FOL_Predicates	% list of symb. guards

Abbreviations

$$\text{Guard} := \bigwedge_{\substack{1\le i\le d\\ \text{sel(i)=pos}}} \text{ctr\_pred(i)} \ \wedge \bigwedge_{\substack{1\le i\le d\\ \text{sel(i)=neg}}} \neg\, \text{ctr\_pred(i)}$$

$$\text{last\_pos} := \begin{cases} \max\{\, i \mid \text{sel(i)=pos} \wedge i\le d\} & \text{, if } \{\, i \mid \text{sel(i)=pos} \wedge i\le d\} \neq \emptyset \\ 0 & \text{, else} \end{cases}$$

$$\text{snd\_last\_pos} := \begin{cases} \max\{\, i \mid \text{sel(i)=pos} \wedge i< d\} & \text{, if } \{\, i \mid \text{sel(i)=pos} \wedge i< d\} \neq \emptyset \\ 0 & \text{, else} \end{cases}$$

Initially $\text{sel}(1) = \text{pos} \ \wedge \ d = 1$

Instructions

$[\!]$ $d < n$
$\longrightarrow$
If satisfiable(Guard)
Then $(\ d' = d + 1 \ \|\ \text{sel}'(i) = \begin{cases} \text{pos} & \text{, } i=d' \\ \text{sel(i)} & \text{, } i\neq d' \end{cases}\ )$
Else (Case

$[\!]$ $\text{sel(d)} = \text{pos} : d'=d+1 \ \|\ \text{sel}'(i) = \begin{cases} \text{neg} & \text{, } i=d \\ \text{pos} & \text{, } i=d' \\ \text{sel(i)} & \text{, else} \end{cases}$

$[\!]$ $\text{sel(d)} = \text{neg} \wedge \text{last\_pos}=0 :$ Halt

$[\!]$ $\text{sel(d)} = \text{neg} \wedge \text{last\_pos}>0 : d'=\text{last\_pos} \ \|\ \text{sel}'(i) = \begin{cases} \text{neg} & \text{, } i=d' \\ \text{sel(i)} & \text{, else} \end{cases}$

End
)

$[\!]$ $d = n$
$\longrightarrow$
If satisfiable(Guard)
Then (symGuardL′= symGuardL $\circ$ Guard ||

Case

$[\!]$ sel(d) = pos : sel′(i) = $\begin{cases} \text{neg} & \text{, i=d} \\ \text{sel(i)} & \text{, else} \end{cases}$

$[\!]$ sel(d) = neg $\wedge$ last_pos=0 : Halt

$[\!]$ sel(d) = neg $\wedge$ last_pos>0 : d′=last_pos || sel′(i) = $\begin{cases} \text{neg} & \text{, i=d}' \\ \text{sel(i)} & \text{, else} \end{cases}$

End

Else (Case

$[\!]$ sel(d) = pos : symGuardL′= symGuardL $\circ$ ($\bigwedge_{\substack{1\le i<n \\ \text{sel(i)=pos}}}$ concrGuard(i) $\wedge$ $\bigwedge_{\substack{1\le i<n \\ \text{sel(i)=neg}}}$ $\neg$ concrGuard(i) $\wedge \neg$ concrGuard(n)) ||

If snd_last_pos=0 Then (Halt)

Else (d′=snd_last_pos ||

sel′(i) = $\begin{cases} \text{neg} & \text{, i=d}' \\ \text{sel(i)} & \text{, else} \end{cases}$)

$[\!]$ sel(d) = neg $\wedge$ last_pos=0 : Halt

$[\!]$ sel(d) = neg $\wedge$ last_pos>0 : d′=last_pos || sel′(i) = $\begin{cases} \text{neg} & \text{, i=d}' \\ \text{sel(i)} & \text{, else} \end{cases}$

End

)

End.

In the program text, $l \circ e$ denotes the list that results from appending the element e to list l. The keyword Halt may be interpreted as a special action that signals the "termination" of a computation. More formally, the occurrence of Halt implies that the block stutters, that is, $\Delta(\text{block}) \Rightarrow \Box$ (Halt $\Rightarrow \bigwedge_{\text{a}\in\text{Ctr(block)}}$ Stutter(a)).

The algorithm generates symbolic guards by a specialized kind of backtracking, that allows to skip the generation of symbolic guards whenever possible (for example, it uses the fact, that if ctr_pred(1) is not satisfiable, then no symbolic guard ctr_pred(1) $\wedge \bigwedge_{\substack{2\le i\le n \\ sel(i)=pos}}$ ctr_pred(i) $\wedge \bigwedge_{\substack{2\le i\le n \\ sel(i)=neg}}$ $\neg$ctr_pred(i) is satisfiable either).

Lemma 5.7.23
If n control predicates pairwise contradict each other, then $\frac{n^2+3*n}{2}$ tests on satisfiability are sufficient to determine the symbolic guards.

Proof:

Let $g_1, \ldots, g_n$ denote the control predicates. Because they exclude each other, the symbolic guards are n predicates $g_i \wedge \bigwedge_{j\neq i} \neg g_j$ (for $1 \le i \le n$) and eventually the predicate $\bigwedge_{1\le i\le n} \neg g_i$.

The first symbolic guard $g_1 \wedge \bigwedge_{1<j\le n} \neg g_j$ is found after the following n tests:

g_1 ,
$g_1 \wedge g_2$,
$g_1 \wedge \neg g_2 \wedge g_3$,
$\vdots$
$g_1 \wedge (\bigwedge_{1<j<n} \neg g_j) \wedge g_n$.

For the second one there are again n tests necessary:

$\neg g_1$,
$\neg g_1 \wedge g_2$,
$\neg g_1 \wedge g_2 \wedge g_3$,
$\vdots$
$\neg g_1 \wedge g_2 \wedge (\bigwedge_{2<j<n} \neg g_j) \wedge g_n$.

To determine the further symbolic guards successively one test less is needed, since the search starts from $\neg g_1 \wedge \neg g_2$, $\neg g_1 \wedge \neg g_2 \wedge \neg g_3$, and so on. Thus, to find, $(\bigwedge_{1\leq j<n} \neg g_j) \wedge g_n$, two tests are necessary:

$\bigwedge_{1\leq j<n} \neg g_j$, and
$(\bigwedge_{1\leq j<n} \neg g_j) \wedge g_n$.

Finally, one more test is required to determine whether $\bigwedge_{1\leq i\leq n} \neg g_i$ is a symbolic guard.

Summing up,

$$n + n + (n-1) + \cdots + 1 \;=\; n + \sum_{1\leq i\leq n} i \;=\; n + \frac{n*(n+1)}{2} \;=\; \frac{n^2+3*n}{2}$$

tests are necessary. ■

5.7.4 Verification Based on Abstract Models

This section is dedicated to a short excursion to verification. It serves to exemplify how abstract BTS can be used for verification purposes and in which way their use is limited.

According to Theorem 5.7.14, there are several ways of checking whether a (temporal) property Ψ holds for a block. Besides proving $\models \Phi(\mathsf{block}) \Rightarrow \Psi$ in temporal logic, it is also possible to show $\mathbf{Traces}(\Sigma(\mathsf{block})) \subseteq \mathbf{Traces}(\Psi)$ for the corresponding BTS. The decision procedures proposed below makes further use of the fact that $\mathbf{Traces}(\Sigma(\mathsf{block})) \subseteq \mathbf{Traces}(\Sigma^s(\mathsf{block}))$. Thus, it is sufficient to decide whether $\mathbf{Traces}(\Sigma^s(\mathsf{block})) \subseteq \mathbf{Traces}(\Psi)$.

The algorithms presented below are based on fix point calculations in a simplified version of $\Sigma^s(\mathsf{block})$: For the invariance properties it is enough to keep a graph where nodes are labeled by one bit (marked or not marked) and the edges are not labeled at all. For leads-to properties one further has to consider fairness as shown below. If $\Sigma^s(\mathsf{block})$ has only finitely many states then both algorithms are decision procedures, that is, they are guaranteed to terminate.

Verification of invariance properties.

To verify safety properties $\Box\psi$ for a state predicate ψ, the following decision procedure may be used:

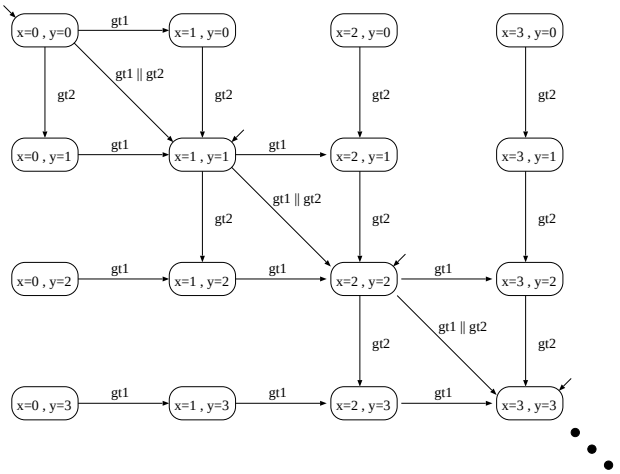
1. Add ψ to the control predicates CP, build the symbolic BTS.
2. Mark all symbolic guards where ψ occurs positive.
3. Calculate the set of reachable states.
4. If all reachable states are marked, then $\mathbf{Traces}(\Sigma^s(\mathsf{block})) \subseteq \mathbf{Traces}(\Box\psi)$.
   If not, $\Box\psi$ may or may not hold. The abstract model has to be refined.

Obviously, a negative decision may already be taken "on the fly" by reaching an unmarked state. To verify $\Box\psi$ for arbitrary transition predicates ψ, one has to mark edges.

As an example consider the block Increment shown in Figure 5.7. To increase readability, in this and the following two figures, let
$\mathsf{gt1} \stackrel{\text{def}}{=} \mathsf{x} \leq \mathsf{y} \wedge \mathsf{x}' = \mathsf{x}+1 \wedge \mathsf{y}' = \mathsf{y}$, $\mathsf{gt2} \stackrel{\text{def}}{=} \mathsf{y} \leq \mathsf{x} \wedge \mathsf{x}' = \mathsf{x} \wedge \mathsf{y}' = \mathsf{y}+1$,
$\mathsf{gt1?} \stackrel{\text{def}}{=} \mathsf{x} < \mathsf{y} \wedge (\mathsf{x}' = \mathsf{x}+1 \vee \mathsf{x}' = \mathsf{x}) \wedge \mathsf{y}' = \mathsf{y}$, $\mathsf{gt2?} \stackrel{\text{def}}{=} \mathsf{y} < \mathsf{x} \wedge \mathsf{x}' = \mathsf{x} \wedge (\mathsf{y}' = \mathsf{y}+1 \vee \mathsf{y}' = \mathsf{y})$,
$\mathsf{gt1} \| \mathsf{gt2} \stackrel{\text{def}}{=} \mathsf{x} = \mathsf{y} \wedge \mathsf{x}' = \mathsf{x}+1 \wedge \mathsf{y}' = \mathsf{y}+1$, and
$(\mathsf{gt1} \| \mathsf{gt2})? \stackrel{\text{def}}{=} \mathsf{x} = \mathsf{y} \wedge (\mathsf{x}' = \mathsf{x}+1 \wedge \mathsf{y}' = \mathsf{y}+1 \vee \mathsf{x}' = \mathsf{x} \wedge \mathsf{y}' = \mathsf{y})$.

For the example Increment, the invariant $\Box\ (\ x = y-1 \vee x = y \vee x = y+1\ )$ holds as may be "seen" in Figure 5.8.

Block Increment
Declarations
Write x,y : Natural
Initially x=y
Transitions
[gt1] x≤y → x′=x+1
[gt2] y≤x → y′=y+1
End



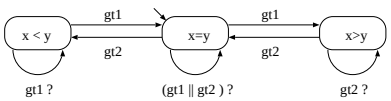


Figure 5.7: On the right, the concrete (on top) and the symbolic BTS corresponding to block Increment are shown.

Verification of leads-to properties.

To verify leads-to properties $\phi \longmapsto \psi$ for state predicate ϕ and ψ, the following decision procedure may be used:

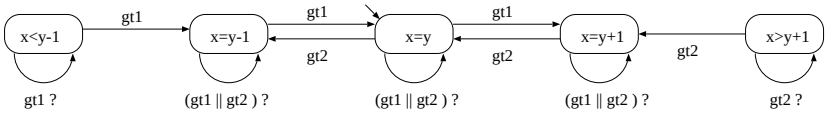


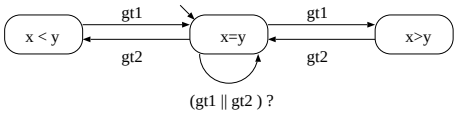
Figure 5.8: The abstract BTS that results from adding $x = y - 1 \vee x = y \vee x = y + 1$ to the set of control predicates.

1. Add ϕ and ψ to the control predicates CP, build the symbolic BTS.
2. Replace the strong fairness conditions by weak fairness conditions using the algorithm presented in Section 4.2.3.
3. Mark all symbolic guards where ψ occurs positive.
4. Label (color) all edges with the weak fairness conditions they take part in
5. Successively
   (1) choose one fairness condition $\mathsf{WF}(G^s, E^s, L^s)$
   (2) mark all nodes $g \in G^s$ where all edges leaving g either
      (1) are edges leading to a marked node, or
      (2) are edges leading to some $g' \in G^s$ but such that $(g, g') \notin E^s$

   until no more nodes can be marked.
6. Determine the set of nodes where ϕ occurs positive. If all nodes in this set are marked, then $\mathbf{Traces}(\Sigma^s(\mathsf{block})) \subseteq \mathbf{Traces}(\phi \longmapsto \psi)$.
   If not, $\phi \longmapsto \psi$ may or may not hold. The abstract model has to be refined or edges $(g, g') \in E^s$ with $g' \in G^s$ have to be removed from E^s due to external knowledge (typically well foundedness of some data type) that guarantees that these edges may not be taken infinitely often without taking an edge leaving G^s.

A more sophisticated algorithm would in addition iterate forward from states marked with ϕ. To allow arbitrary transition predicates ϕ and ψ, one also has to mark the edges.

In the example, this decision procedure does not yield to a direct result for the property $\mathsf{true} \longmapsto x = y$: in the symbolic BTS presented in Figure 5.7, the self-loops in the abstract states $x < y$ and $x > y$ foil the marking of these states. However, these self-loops can be removed, because it is not possible to increment (decrement) x permanently in state $x < y$ ($x > y$). The resulting BTS then is given by



In this BTS both states $x < y$ and $x > y$ are marked in the first iteration of step 5 in the algorithm. Thus the property $\mathsf{true} \longmapsto x = y$ holds.

5.8 Modules

In first approximation, modules consist of a set of blocks that share the same declarations while they control disjoint variables and actions. On the level of modules, hiding is introduced.

Definition 5.8.1 (Modules)

A tuple $<$ Mod, Types, Params, $\mathsf{Decl}_{\mathsf{Mod}}$, Init, Transitions, Fair, Blocks $>$ *is called module* **iff**

- $<$ Bl^0, Types, Params, $\mathsf{Decl}_{\mathsf{Mod}}$, Init, Transitions, Fair $>$ *is a block, called the main block of the module, and*
- Blocks *is a finite set of blocks with each block* Bl *being enriched with a (possibly empty) list of substitutions such that*

  [Subst1] *all parameters of* Bl *get replaced by expressions containing only constants or module parameters of appropriate type, and*

  [Subst2] *any identifier occurring in* Bl *may be renamed either by a fresh identifier or by identifiers from* $\mathsf{Decl}_{\mathsf{Mod}}$ *given both identifiers agree in type and environment class.*

  These substitutions are carried out simultaneously in a preprocessing step, that is, before determing Vars, Acts, *etc. or extracting the transition relations of the blocks.*

Let Decls *denote the union of* $\mathsf{Decl}_{\mathsf{Mod}}$ *with all declarations made in any of the blocks after all substitutions have been carried out.* Vars *and* Acts *then denote the sets of all declared variables and actions respectively.*
Finally, in order to avoid duplicate declarations, variables and actions need to *be declared only within the block which controls them. This convention makes it necessary to add missing declarations to the blocks. That is,* Decls(Bl) *has to be replaced by* Decls(Bl) $\cup \{<$ a, type, env_class $> \in$ Decls(Mod) $\mid$ a $\notin$ Decls(Bl) *and* a *occurs in* Bl$\}$ *for all blocks.*

Furthermore and wlog, let Blocks $= \{\mathsf{Bl}^1, \ldots, \mathsf{Bl}^N\}$ *denote the blocks, again after all substitutions have been carried out. Thus,* $\{\mathsf{Bl}^0, \ldots, \mathsf{Bl}^N\}$ *denotes all blocks.*

The set of controlled actions and variables of the whole module is defined as the union of the corresponding sets of the blocks:

$$\mathrm{Ctr}(\mathsf{Mod}) \stackrel{def}{=} \bigcup_{0 \le i \le N} \mathrm{Ctr}(\mathsf{Bl}^i)$$

Then the following consistency conditions must be fulfilled:

1. *No identifier occurs several times in* Decls *with different type or environment class.*
2. *The control sets of the blocks are pairwise disjoint:*
   [ExclCtr] $\mathrm{Ctr}(\mathsf{Bl}^i) \cap \mathrm{Ctr}(\mathsf{Bl}^j) = \emptyset$ *for all* $i \neq j$
3. *The specification variable lists occurring in the triggered transitions are pairwise disjoint.*
4. *Pairs of triggered transitions are not enabled simultaneously (with regard to their assumptions) or one of the delays is more general:*

   For any pair tt_1,tt_2 *of triggered transitions occurring in different blocks*

[TT_Consistency_Mod]

$$[\![\, \mathsf{ev_i} \wedge \mathsf{ev_j} \;\Rightarrow\; \neg(\mathsf{assume_i} \wedge \mathsf{assume_j}) \;\vee\; (\mathsf{delay_i} \Leftarrow \mathsf{delay_j}) \;\vee\; (\mathsf{delay_i} \Rightarrow \mathsf{delay_j}) \,]\!]$$

5. *All fairness conditions are executable: Let* $<$ id, F, fg, fcmd $>$ *be some arbitrary fairness condition occurring in one of the blocks. Then the following has to hold:*

   [Fair_Module]

   $$[\![\, \mathsf{fg} \;\Rightarrow\; \exists_{\mathrm{Ctr'(Mod)}}\, (\, \mathrm{Trl}(\mathsf{fcmd}) \wedge \bigwedge_{0 \le \mathsf{i} \le \mathsf{N}} \delta(\mathsf{Bl^i})\,)\,]\!]$$

6. *The safety commitments are fulfilled:*
   for all transitions $<$ tt, s, assume, delay, out_ev, cmd $>$ *occurring in one of the blocks* $\mathsf{Bl^k}$,

   [Commit_TL] $\quad \models \bigwedge_{0 \le \mathsf{i} \le \mathsf{N}} \Delta^{-\mathsf{Com}}(\mathsf{Bl^i}) \Rightarrow \Box\, \forall_{\mathsf{s}}\, (\mathsf{out\_ev} \Rightarrow \mathsf{assume})$

   where $\Delta^{-\mathsf{Com}}(\mathsf{Bl^k})$ *is obtained from* $\Delta(\mathsf{Bl^k})$ *by replacing the occurrence of all commitments* assume *by* true, *that is,*

   $$\forall_{\mathsf{s}}\, (\; \mathsf{out\_ev} \Rightarrow \mathsf{delay} \wedge \mathsf{assume} \wedge \mathrm{Trl}(\mathsf{cmd})\;)$$

   reduces to

   $$\forall_{\mathsf{s}}\, (\; \mathsf{out\_ev} \Rightarrow \mathsf{delay} \wedge \mathrm{Trl}(\mathsf{cmd})\;)$$

   in the definition of $\delta_{env}(\mathsf{Bl^k})$.

   Often, temporal reasoning is not necessary. A sufficient first-order logic criteria is given by

   [Commit_FOL] $\quad [\![\bigwedge_{0 \le i \le N} \delta^{-\mathsf{Com}}(\mathsf{Bl^i}) \Rightarrow\; \forall_{\mathsf{s}}\, (\mathsf{out\_ev} \Rightarrow \mathsf{assume})\,]\!]$

7. *The delays are satisfied:*

   [Delay1] $\quad [\![\; \forall_{\mathsf{s}}\, (\; \mathsf{assume} \wedge \neg\mathsf{delay} \Rightarrow \exists_{\mathrm{Ctr'(Mod)}}\, (\; \bigwedge_{0 \le \mathsf{i} \le \mathsf{N}} \delta(\mathsf{Bl^i}) \wedge \bigwedge_{\{a \,\in\, \mathrm{Ctr}(Mod) \;\mid\; a \,\in\, \mathsf{vVars} \cup \mathsf{vActs}\}} \mathsf{Stutter(a)}\;)\;)\,]\!]$

   [Delay2] $\quad \models \bigwedge_{0 \le \mathsf{i} \le \mathsf{N}} \Phi(\mathsf{Bl^i}) \Rightarrow\; \forall_{\mathsf{s}}\, \Box\, \Diamond\, (\; \neg\mathsf{assume} \vee \mathsf{delay}\;)$

   for all triggered transitions $<$ tt, s, assume, delay, ev, cmd $>$ *occurring in one of the blocks.*

Remark 1. In the premises of the consistency criteria dealing with the safety commitments and with delays, all blocks appear. Mostly, it is possible to consider only the blocks that control the Out-event or the delayed event. In case of the safety commitments one then proves

$$\models \bigwedge_{\mathsf{j} \in \mathsf{J}} \Delta^{-\mathsf{Com}}(\mathsf{Bl^j}) \Rightarrow \Box\, \forall_{\mathsf{s}}\, (\mathsf{out\_ev} \Rightarrow \mathsf{assume})$$

where $J \stackrel{\mathrm{def}}{=} \{j \mid 0 \le j \le N$ and there is some action $A \in \mathrm{Ctr}(Bl^j)$ that occurs in out_ev $\}$. The syntactically shortest proof obligations for the safety commitments are obtained by the following (sufficient) criteria:

[Commit_FOL2]

$$[\![\; (\; \bigwedge_{\mathsf{A} \in \mathsf{FAct(out\_ev)}} \quad \bigvee_{\substack{<\, \mathsf{gt},\, \mathsf{guard},\, \mathsf{cmd} \,> \\ \mathrm{Ctr(cmd)} \cap \mathsf{FAct(out\_ev)} \neq \emptyset}} \mathsf{guard} \wedge \mathrm{Trl}(\mathsf{cmd})\;) \Rightarrow\; \forall_{\mathsf{s}}\, (\mathsf{out\_ev} \Rightarrow \mathsf{assume})\,]\!]$$

That is, in contrast to [Commit_FOL], only those guarded transitions are considered that emit actions occurring in out_ev.

In case of the delays, $\bigwedge_{0 \le i \le N} \Phi(\mathsf{Bl}^i)$ can often be replaced by $\Phi(\mathsf{Bl})$ for the block where tt occurs. Furthermore, the delay condition is trivially fulfilled for triggered transitions dealing with Out-events, because they may not be delayed (delay = true).

REMARK 2. The proofs concerned with the commitments can further be simplified by "adding" the commitments that have already been proven successively to the premise of the consistency conditions. That is, $\Delta^{-\mathsf{Com}}$ may be replaced by a stronger predicate by only "replacing" only those commitments that still have to be proven.

Lemma 5.8.2

Let $\Delta^{-\mathsf{Com}}(\mathsf{Bl})$ *denote the formula that is obtained from* $\Delta(\mathsf{block})$ *by removing (i.e. replacing by* true*) all commitments.*

If [Commit_TL] is fulfilled for all commitments, then

$$\models \Big(\bigwedge_{0 \le i \le N} \Delta(\mathsf{Bl}^i)\Big) \;\Leftrightarrow\; \Big(\bigwedge_{0 \le i \le N} \Delta^{-\mathsf{Com}}(\mathsf{Bl}^i) \wedge \bigwedge_{< \mathsf{tt},\, \mathsf{s},\, \mathsf{assume},\, \mathsf{delay},\, \mathsf{out\_ev},\, \mathsf{cmd} > \in \bigcup \mathsf{Transitions}} \Box\, \forall_s \,(\, \mathsf{out\_ev} \Rightarrow \mathsf{assume}\,)\Big)$$

$$\Leftrightarrow\; \Big(\bigwedge_{0 \le i \le N} \Delta^{-\mathsf{Com}}(\mathsf{Bl}^i)\Big)$$

Proof:

The first equivalence follows from the definitions of $\Delta(\mathsf{Bl}^i)$ and $\Delta^{-\mathsf{Com}}(\mathsf{Bl}^i)$, the second equivalence is due to [Commit_TL].

■

Notation:

Module Mod

Types

T := type_definition

Parameters

param : sort(param)

Declarations

class(a) a : sort(a)

Abbreviations

Abbrev := expr

Initially init

Transitions

[gt] guard → cmd

$[\mathsf{tt}]_{\mathsf{s}}$ {assume} ⟨delay⟩ ev =▷cmd

Fairness

[wf] WF(fg,fcmd)

[sf] SF(fg,fcmd)

Include Block $\mathsf{Bl1}_{[\mathsf{id}_1 \leftarrow \mathsf{t}_1, \ldots, \mathsf{id}_k \leftarrow \mathsf{t}_k]}$

Include Inverted Block $\mathsf{Bl2}_{[\mathsf{id}_1 \leftarrow \mathsf{t}_1, \ldots, \mathsf{id}_l \leftarrow \mathsf{t}_l]}$

(* if Bl2 is an interface *)

Block Bl3

⋮

End

End(* Module *)

For interfaces that are included into the module, it is possible to specify them as Inverted. This means, that after all substitutions have been carried out, the environment classes of all declared visible actions get inverted: In-actions become Out-actions and vice versa.

Example: Peterson's mutual exclusion algorithm

The following module describes a way to model tokens without using global variables. For simplicity, it is assumed that the token may be set by only two In-actions Set_0 and Set_1.

Module Token

Declarations

Variables

Write	token	: [0..1]

Actions

In	Set_0, Set_1	: ()

Transitions

[set_0] $\text{Set\_0} \ \wedge \ \neg\text{Set\_1} \ \Rightarrow \ \text{token}' = 0$

[set_1] $\neg\text{Set\_0} \ \wedge \ \text{Set\_1} \ \Rightarrow \ \text{token}' = 1$

[both] $\text{Set\_0} \ \wedge \ \text{Set\_1} \ \Rightarrow \ \text{token}' = \text{token}'$

End.

Quite straightforward, if exactly one In-action occurs then the value of token is set accordingly. More interesting is transition [both]. If both In-actions occur simultaneously, the value of token may be set to any value ($\text{token}' = \text{token}'$ is equivalent to $\text{token}' \in \{0, 1\}$). Because there are no two events that may be fulfilled simultaneously, there is no need for assumptions.

As a second simple example consider the following client that uses token to gain access to a critical section:

Module Client

Parameters

myToken : [0..1]

Declarations

Variables

Read	token	: [0..1]	
Read	req_1	: Boolean	
Write	req_0, cs_0	: Boolean	Init false
Local	pc	: [0..3]	

Actions

Out	Set_1	: ()

Transitions

$[\text{request}]\ \ \text{pc} = 0 \ \longrightarrow\ \text{req}' \ ||\ \ \text{pc}' = 1$

$[\text{set\_token}]\ \ \text{pc} = 1 \ \longrightarrow\ \text{Set\_1} \ ||\ \ \text{pc}' = 2$

$[\text{enter1}]\ \ \text{pc} = 2 \ \wedge\ \neg\text{req\_1} \ \longrightarrow\ \text{cs\_0}' \ ||\ \ \text{pc}' = 3$

$[\text{enter2}]\ \ \text{pc} = 2 \ \wedge\ \text{token} = \text{myToken} \ \longrightarrow\ \text{cs\_0}' \ ||\ \ \text{pc}' = 3$

$[\text{leave}]\ \ \text{pc} = 3 \ \longrightarrow\ \neg\text{req\_0}' \ ||\ \ \neg\text{cs\_0}' \ ||\ \ \text{pc}' = 0$

End.

Intuitively, Peterson's algorithm now works like this: Firstly, the client makes a request (transition [request]), then the client offers some other client the right of way by sending Set_1. Then the client module checks whether it has right of way ([enter1]) or whether the other client has not even made a request ([enter2]) and enters its critical section. Finally, executing [leave], the client leaves the critical section again.

□

Definition 5.8.3 (Translation to Logic and concrete BTS)

Assuming the notation of Definition 5.8.1,

$$\delta(\mathsf{Mod}) \stackrel{def}{=} \bigwedge_{0\le i\le N} \delta(\mathsf{Bl}^{\mathsf{i}}) \qquad \delta^{vis}(\mathsf{Mod}) \stackrel{def}{=} \exists_{\mathsf{hVars,hActs,hVars'}}\, \delta(\mathsf{Mod})$$

$$\Delta(\mathsf{Mod}) \stackrel{def}{=} \bigwedge_{0\le i\le N} \Delta(\mathsf{Bl}^{\mathsf{i}}) \qquad \Delta^{vis}(\mathsf{Mod}) \stackrel{def}{=} \exists_{\mathsf{hVars,hActs}}\, \Delta(\mathsf{Mod})$$

$$\Phi(\mathsf{Mod}) \stackrel{def}{=} \bigwedge_{0\le i\le N} \Phi(\mathsf{Bl}^{\mathsf{i}}) \qquad \Phi^{vis}(\mathsf{Mod}) \stackrel{def}{=} \exists_{\mathsf{hVars,hActs}}\, \Phi(\mathsf{Mod})$$

$$\Sigma(\mathsf{Mod}) \stackrel{def}{=} \|_{0\le i\le N} \Sigma(\mathsf{Bl}^{\mathsf{i}}) \qquad \Sigma^{vis}(\mathsf{Mod}) \stackrel{def}{=} \Sigma(\mathsf{Mod})\big|_{\mathsf{(vVars,vActs)}}$$

where the definition of $\Sigma(\mathsf{Mod})$ *is based on canonical embeddings of the Boolean algebras underlying* $\Sigma(\mathsf{Bl}^{\mathsf{i}})$ *and where* $|_{\mathsf{(vVars,vActs)}}$ *denotes the canonical projection*

$$|_{\mathsf{(vVars,vActs)}} \ :\ \mathcal{P}\Big(\prod_{x\in\mathsf{Vars}}\mathcal{D}(x) \times \prod_{A\in\mathsf{Acts}}\mathcal{D}_\perp(A) \times \prod_{x'\in\mathsf{Vars}}\mathcal{D}(x')\Big) \twoheadrightarrow \mathcal{P}\Big(\prod_{x\in\mathsf{vVars}}\mathcal{D}(x) \times \prod_{A\in\mathsf{vActs}}\mathcal{D}_\perp(A) \times \prod_{x'\in\mathsf{vVars}}\mathcal{D}(x')\Big).$$

Examples :

The translation to logic for module Token is given by

$$\Phi^{\mathsf{vis}}(\mathsf{Token}) \ =\ \Phi(\mathsf{Token}) \ =\ \Box\ (\ \mathsf{token}' \neq \mathsf{token} \Rightarrow \begin{array}{ll} & Set\_0 \wedge \neg Set\_1 \wedge token' = 0 \\ \vee & \neg Set\_0 \wedge Set\_1 \wedge token' = 1 \\ \vee & Set\_0 \wedge Set\_1 \wedge token' = token' \end{array} \ )$$

For module Client, firstly let

$\mathsf{request} \stackrel{\text{def}}{=} \mathsf{pc} = 0 \wedge \mathsf{req}' \wedge \mathsf{pc}' = 1$

$\mathsf{set\_token} \stackrel{\text{def}}{=} \mathsf{pc} = 1 \wedge \mathsf{Set\_1} \wedge \mathsf{pc}' = 2$

$\mathsf{enter1} \stackrel{\text{def}}{=} \mathsf{pc} = 2 \wedge \neg\mathsf{req\_1} \wedge \mathsf{cs\_0}' \wedge \mathsf{pc}' = 3$

$\mathsf{enter2} \stackrel{\text{def}}{=} \mathsf{pc} = 2 \wedge \mathsf{token} = \mathsf{myToken} \wedge \mathsf{cs\_0}' \wedge \mathsf{pc}' = 3$

$\mathsf{leave} \stackrel{\text{def}}{=} \mathsf{pc} = 3 \wedge \neg\mathsf{req\_0}' \wedge \neg\mathsf{cs\_0}' \wedge \mathsf{pc}' = 0$

$\mathsf{any\_guard} \stackrel{\text{def}}{=} \mathsf{pc} = 0 \vee \mathsf{pc} = 1 \vee \mathsf{pc} = 2 \wedge \neg \mathsf{req\_1} \vee \mathsf{pc} = 2 \wedge \mathsf{token} = \mathsf{myToken} \vee \mathsf{pc} = 3$

$\mathsf{any\_transition} \stackrel{\text{def}}{=} \mathsf{request} \vee \mathsf{set\_token} \vee \mathsf{enter1} \vee \mathsf{enter2} \vee \mathsf{leave}$

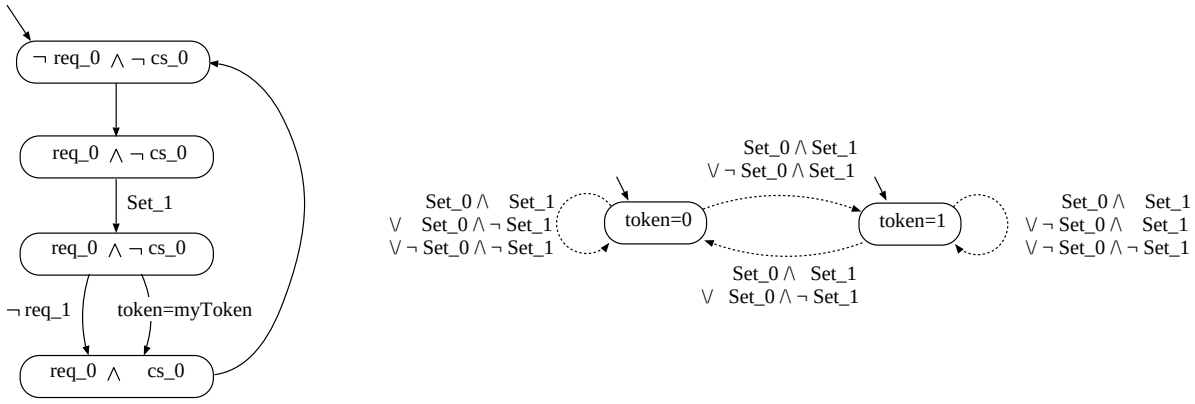
Then,

$$\Phi(\mathsf{Client}) \;=\; \Box\,\Big(\; \begin{array}{ll} & (\; cs\_0' \neq cs\_0 \Rightarrow enter1 \vee enter2 \vee leave\;) \\ \wedge & (\; req\_0' \neq req\_0 \Rightarrow request \vee leave\;) \\ \wedge & (\; pc' \neq pc \Rightarrow\; any\_transition\;) \\ \wedge & (\; Set\_1pc \Rightarrow\; set\_token\;) \end{array} \;\Big) \\ \wedge\; \mathcal{WF}(\mathsf{any\_guard}, \mathsf{any\_transition})$$

and

$$\Phi^{\mathsf{vis}}(\mathsf{Client}) \;=\; \exists_{\,\mathsf{pc}}\Phi(\mathsf{Client})$$

The corresponding BTS are given in their pictorial representation:



In the BTS representing module Token, note that the self-loop in state $\mathsf{token} = 0$ could take place due to transition [set_0] or due to transition [both] or due to stuttering.

□

REMARK 1. $\Sigma(\mathsf{Mod})$ is well defined:

For simplicity, it is assumed that all fairness conditions are weak fairness conditions (compare Theorem 4.2.7). These may be rewritten as Büchi conditions (Corollary 4.2.6). According to Definition 4.2.21, it remains to show that [St_Independence] holds for the images of the BTS $\Sigma(Bl^n) \stackrel{\text{def}}{=} (V^n, I^n, \mathcal{B}^n, st^n, M^n, \mathcal{F}^n)$ under the canonical embeddings φ_n:
The blocks are stutter independent **iff** for any set of edges $e^n \in V^n \times V^n$ such that $\emptyset \neq \varphi_n(st^n) \cap \varphi_n(M^n(e^n))$, it holds that $\emptyset \neq \bigcap_{0 \leq n \leq N} \varphi_n(st^n) \cap \varphi_n(M^n(e^n))$:

If $(\xi_n, \alpha_n, \xi_n') \in \varphi_n(st^n) \cap \varphi_n(M^n(v^n, w^n))$, then by definition of $\varphi_n(st^n)$ it follows that $\alpha_n(A) = \bot$ for all $A \in \mathsf{Acts}(\mathsf{Bl}^n)$ and $\xi_n(x) = {\xi_n}'(x)$ for all $x \in \mathsf{Vars}(\mathsf{Bl}^n)$. Furthermore, by definition of $\varphi_n(M^n)$, it follows that $v^n = w^n$.

Defining $\xi(x) = \begin{cases} v^n(x) & , x \in \mathrm{Ctr}_V(\mathsf{Bl}^n) \\ \text{arb.} & , x \in \mathsf{Vars} - \bigcup_{0 \leq n \leq N} \mathrm{Ctr}_V(\mathsf{Bl}^n) \end{cases}$, $\xi' \stackrel{\text{def}}{=} \xi$ and $\alpha = \tau$,

$(\xi, \alpha, \xi') \in \bigcap_{0 \leq n \leq N} \varphi_n(st^n)$ and by Lemma 5.7.6 it follows that $(\xi, \alpha, \xi') \in \varphi_n(M^n(v^n, v^n))$ for all $0 \leq n \leq N$ and thus $(\xi, \alpha, \xi') \in \bigcap_{0 \leq n \leq N} \varphi_n(st^n) \cap \varphi_n(M^n(e^n))$ as required.

REMARK 2. The explicit representation of $\Sigma(\mathsf{Mod})$ agrees with the one given for blocks in Definition 5.7.11 (one only has to replace "block" by "Mod").

For example,

$$
\begin{aligned}
st(\mathsf{Mod}) &\stackrel{\text{def}}{=} \bigcap_{0\le n\le N} \varphi_n(st(\mathsf{Bl}^n)) \\
&= \bigcap_{0\le n\le N} \varphi_n(\{(\chi,\tau,\chi') \in \mathcal{L}(\mathsf{Bl}^n) \mid \chi(x) = \chi'(x') \text{ for all } x \in \mathsf{Vars}(\mathsf{Bl}^n)\}) \\
&= \bigcap_{0\le n\le N} \{(\chi,\alpha,\chi') \in \mathcal{L}(\mathsf{Mod}) \mid \chi(x) = \chi'(x') \text{ for all } x \in \mathsf{Vars}(\mathsf{Bl}^n) \text{ and} \\
&\qquad\qquad \alpha(A) = \bot \text{ for all } A \in \mathsf{Acts}(\mathsf{Bl}^n)\} \\
&= \{(\chi,\tau,\chi') \in \mathcal{L}(\mathsf{Mod}) \mid \chi(x) = \chi'(x') \text{ for all } x \in \mathsf{Vars}(\mathsf{Mod})\} \ .
\end{aligned}
$$

REMARK 3. The stuttering element of $\Sigma^{vis}(\mathsf{Mod})$ is given by

$$
\begin{aligned}
st^{vis}(\mathsf{Mod}) = \{(\chi,\alpha,\chi') \in \mathcal{L}(\mathsf{Mod})\big|_{(\mathsf{vVars},\mathsf{vActs})} \mid \chi(x) = \chi'(x') \text{ for all } x \in \mathsf{vVars}(\mathsf{Mod}) \text{ and} \\
\alpha(A) = \bot \text{ for all } A \in \mathsf{vActs}(\mathsf{Mod})\} \quad .
\end{aligned}
$$

That is, the values of local variables may be changed while stuttering. On the other hand, assumptions are preserved by visible stuttering steps because they are based only on visible and history variables (recall that the latter ones may change their value only if a visible event occurs).

Having introduced hiding, it is reasonable to also distinguish local and visible fairness conditions. Because of the general definition of fairness conditions, it is possible that although a fairness condition only names local variables, there may not be a local step fulfilling the fairness condition. For example, consider the module

Module Side-effect

Declarations

Variables

Local l : Natural

Actions

Out A : ()

Transitions

[] true $\longrightarrow$ l′= l+1 || A

Fairness

[wf] WF(true,l′= l+1)

End.

In this example, the only possibility to do a fair step as a side effect results in emitting the visible action A. Therefore, the following definition is necessary:

Definition 5.8.4 (Local Fairness Conditions)

A fairness condition $<$ id, F, fg, fcmd $>$ *occurring in module* Mod *is called local* **iff**

$$[\![\mathsf{fg} \;\Rightarrow\; \exists_{\mathrm{Ctr}'(\mathsf{Mod})} (\mathrm{Trl}(\mathsf{fcmd}) \wedge \delta(\mathsf{Mod}) \wedge \bigwedge_{\mathsf{a} \in \mathsf{vVars} \cup \mathsf{vActs}} \mathsf{Stutter(a)})]\!].$$

All other fairness conditions are called visible.

Again, one now has to show that both representations yield the same set of traces and that there is at least one trace. Firstly, embedding the declarations of the blocks into the declarations of the module has the same effect on the sets of traces in both the logical and the automata view:

Lemma 5.8.5 (Embedding)
Let

$$\varphi_i : \mathcal{P}(\prod_{x \in \mathsf{Vars}(\mathsf{Bl}^{\mathsf{i}})} \mathcal{D}(x) \times \prod_{A \in \mathsf{Acts}(\mathsf{Bl}^{\mathsf{i}})} \mathcal{D}_\perp(A) \times \prod_{x' \in \mathsf{Vars}'(\mathsf{Bl}^{\mathsf{i}})} \mathcal{D}(x')) \hookrightarrow \mathcal{P}(\prod_{x \in \mathsf{Vars}} \mathcal{D}(x) \times \prod_{A \in \mathsf{Acts}} \mathcal{D}_\perp(A) \times \prod_{x' \in \mathsf{Vars}'} \mathcal{D}(x'))$$

denote the canonical embedding of the declarations of block i *into the declarations of the module. Then*

$$\mathbf{Traces}_{\mathsf{Vars}(\mathsf{Bl}^{\mathsf{i}}),\mathsf{Acts}(\mathsf{Bl}^{\mathsf{i}})}(\Sigma(\mathsf{Bl}^{\mathsf{i}})) = \mathbf{Traces}_{\mathsf{Vars}(\mathsf{Bl}^{\mathsf{i}}),\mathsf{Acts}(\mathsf{Bl}^{\mathsf{i}})}(\Phi(\mathsf{Bl}^{\mathsf{i}}))$$

implies $\mathbf{Traces}_{\mathsf{Vars},\mathsf{Acts}}(\varphi_i \Sigma(\mathsf{Bl}^{\mathsf{i}})) = \mathbf{Traces}_{\mathsf{Vars},\mathsf{Acts}}(\Phi(\mathsf{Bl}^{\mathsf{i}}))$.

Proof:

$\sigma \in \mathbf{Traces}_{\mathsf{Vars},\mathsf{Acts}}(\varphi_i \Sigma(\mathsf{Bl}^{\mathsf{i}}))$
iff /* Lemma 4.2.20 */
$\sigma|_{(\mathsf{Vars}(\mathsf{Bl}^{\mathsf{i}}),\mathsf{Acts}(\mathsf{Bl}^{\mathsf{i}}))} \in \mathbf{Traces}_{(\mathsf{Vars}(\mathsf{Bl}^{\mathsf{i}}),\mathsf{Acts}(\mathsf{Bl}^{\mathsf{i}}))}(\Sigma(\mathsf{Bl}^{\mathsf{i}}))$
iff /* Premise */
$\sigma|_{(\mathsf{Vars}(\mathsf{Bl}^{\mathsf{i}}),\mathsf{Acts}(\mathsf{Bl}^{\mathsf{i}}))} \in \mathbf{Traces}_{(\mathsf{Vars}(\mathsf{Bl}^{\mathsf{i}}),\mathsf{Acts}(\mathsf{Bl}^{\mathsf{i}}))}(\Phi(\mathsf{Bl}^{\mathsf{i}}))$
iff /* Definition 3.2.5 */
$\sigma \in \mathbf{Traces}_{\mathsf{Vars},\mathsf{Acts}}(\Phi(\mathsf{Bl}^{\mathsf{i}}))$

■

Therefore, applying Theorem 5.7.14 (stating that $\mathbf{Traces}_{\mathsf{Vars}(\mathsf{Bl}^{\mathsf{i}}),\mathsf{Acts}(\mathsf{Bl}^{\mathsf{i}})}(\Sigma(\mathsf{Bl}^{\mathsf{i}})) = \mathbf{Traces}_{\mathsf{Vars}(\mathsf{Bl}^{\mathsf{i}}),\mathsf{Acts}(\mathsf{Bl}^{\mathsf{i}})}(\Phi(\mathsf{Bl}^{\mathsf{i}}))$) to all block results in

Corollary 5.8.6

$$\bigcap_{0 \le i \le N} \mathbf{Traces}_{\mathsf{Vars},\mathsf{Acts}}(\varphi_i \Sigma(\mathsf{Bl}^{\mathsf{i}})) = \bigcap_{0 \le i \le N} \mathbf{Traces}_{\mathsf{Vars},\mathsf{Acts}}(\Phi(\mathsf{Bl}^{\mathsf{i}}))$$

The following lemma dealing with hiding will be helpful in the proof of Theorem 5.8.8.

Lemma 5.8.7 (Hiding)

1. $\mathbf{Traces}(\Sigma) \neq \emptyset$ *implies* $\mathbf{Traces}(\Sigma|_{(\mathsf{vVars},\mathsf{vActs})}) \neq \emptyset$
2. *If* $\mathbf{Traces}(\Sigma) = \mathbf{Traces}(\Phi)$
   then $\mathbf{Traces}(\Sigma|_{(\mathsf{vVars},\mathsf{Acts})}) = \mathbf{Traces}_{\mathsf{vVars},\mathsf{vActs}}(\exists_{\mathsf{hVars},\mathsf{hActs}}\ \Phi)$

Proof:

1. Immediate from Lemma 4.2.24 in Chapter 4.

2. $\sigma \in \mathbf{Traces}(\Sigma|_{(\mathsf{vVars,vActs})})$
   iff /* Lemma 4.2.24 */
   $\sigma \in \mathbf{Traces}(\Sigma)|_{(\mathsf{vVars,vActs})}$
   iff /* Definition 4.2.5 */
   exists $\tilde{\sigma} \in \mathbf{Traces}(\Sigma)$ such that $\tilde{\sigma}|_{(\mathsf{vVars,vActs})} = \sigma$
   iff /* Premise */
   exists $\tilde{\sigma} \in \mathbf{Traces}(\Phi)$ such that $\tilde{\sigma}|_{(\mathsf{vVars,vActs})} = \sigma$
   iff /* Definition 3.2.3, Definition 3.2.5 */
   $\sigma \in \mathbf{Traces}_{\mathsf{vVars,vActs}}(\exists_{\mathsf{hVars,hActs}}\ \Phi)$

■

Theorem 5.8.8

1. $\mathbf{Traces}(\mathsf{Mod}) \stackrel{def}{=} \mathbf{Traces}(\Sigma(\mathsf{Mod})) = \mathbf{Traces}(\Phi(\mathsf{Mod}))$
2. $\mathbf{Traces}(\mathsf{Mod}^{vis}) \stackrel{def}{=} \mathbf{Traces}(\Sigma^{vis}(Mod)) = \mathbf{Traces}(\Phi^{vis}(Mod))$
3. $\mathbf{Traces}(\mathsf{Mod}) \neq \emptyset$
4. $\mathbf{Traces}(\mathsf{Mod}^{vis}) \neq \emptyset$

Proof:

1. $$\mathbf{Traces}(\|_{0\leq i\leq N}\Sigma(\mathsf{Bl}^{\mathsf{i}})) \stackrel{Theorem 4.2.10}{=} \bigcap_{0\leq i\leq N} \mathbf{Traces}(\varphi_i\Sigma(\mathsf{Bl}^{\mathsf{i}})) \stackrel{Corollary 5.8.6}{=} \bigcap_{0\leq i\leq N} \mathbf{Traces}(\Phi(\mathsf{Bl}^{\mathsf{i}})) \stackrel{Lemma 3.2.6}{=} \mathbf{Traces}(\bigwedge_{0\leq i\leq N} \Phi(\mathsf{Bl}^{\mathsf{i}})).$$

2. Follows from 1 by applying 5.8.7(2).

3. Firstly, in Lemma 5.8.2, it was shown that
   $\mathbf{Traces}(\bigwedge_{0\leq i\leq N} \Delta(\mathsf{Bl}^{\mathsf{i}})) = \mathbf{Traces}(\bigwedge_{0\leq i\leq N} \Delta^{-\mathsf{Com}}(\mathsf{Bl}^{\mathsf{i}}))$.
   Adding fairness on both sides results in
   $\mathbf{Traces}(\bigwedge_{0\leq i\leq N} \Phi(\mathsf{Bl}^{\mathsf{i}})) = \mathbf{Traces}(\bigwedge_{0\leq i\leq N} \Phi^{-\mathsf{Com}}(\mathsf{Bl}^{\mathsf{i}}))$.
   Therefore, replacing all commitments by true does not influence the set of traces.

   Now, the proof of the theorem can be traced back to Theorem 4.2.17. [4]Therefore, let $\Sigma(Bl^n) \stackrel{\text{def}}{=} (V^n, I^n, \mathcal{B}^n, st^n, M^n, \mathcal{F}^n)$. Because no own fairness condition is delayed (recall that $\mathsf{delay}{=}\mathsf{true}$ in all triggered transitions dealing with Out-events), it is sufficient to show input enabledness, stutter independence and fairness independence:

(a) Any block Bl^n of the module is input enabled with respect to the fairness conditions of any other block Bl^f: According to definition 4.2.12, it has to be shown that for any fairness condition $\mathsf{WF}(G, E, L) \in \mathcal{F}^k$, any state $w \in G$ and any state $v \in V^n$ (where $n \neq k$) there are some $v' \in V^n$, $w' \in V^k$ and a valuation $\mu \in \mathcal{L}$ such that $(w, w') \in E$ and $\mu \in \varphi_k(L(w, w')) \cap \varphi_n(M^n(v, v'))$.

If $w \in G$ then by definition $[\![\mathsf{fg}]\!]^w$ and by [Fair_Module], it follows that there exists some μ such that

$$[\![\mathsf{fg} \ \wedge \ \mathrm{Trl}(\mathsf{fcmd}) \wedge \bigwedge_{0 \leq \mathsf{i} \leq \mathsf{N}} \delta(\mathsf{Bl}^{\mathsf{i}})]\!]^\mu$$

Thus, it holds that $[\![\mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{Bl}^k)]\!]^\mu$ and, by construction, $(w, w') \in E$. Besides, $[\![\delta(\mathsf{Bl}^n)]\!]^\mu$ and therefore $\mu \in \varphi_k(L(w, w')) \cap \varphi_n(M^n(v, v'))$ holds for $v'(x) \stackrel{\text{def}}{=} \mu(x)$ for all $x \in \mathrm{Ctr}(\mathsf{Bl}^n)$.

(b) That the blocks are stutter independent was already shown in the remarks following Definition 5.8.3.

(c) Remains fairness independence. Let $v^n \in V^n$ and in addition for some fixed but arbitrary k let $v^k \in G$ for some $\mathsf{WF}(G, E, L) \in \mathcal{F}(\mathsf{Bl}^k)$.
Then, it has to be shown that if for all $n \neq k$ there are $w^n \in V^n$ and $w^k \in V^k$ such that $\emptyset \neq \varphi_k(L(v^k, w^k)) \cap \varphi_n(M^n(v^n, w^n))$ then it holds that

(*) $\emptyset \neq \varphi_k(L(v^k, \tilde{w}^k)) \cap \bigcap_{0 \leq n \leq N} \varphi_n(M^n(v^n, \tilde{w}^n))$ for some $\tilde{w}^n \in V_n$.

This means that when all blocks are input enabled wrt a certain fairness condition, then they can do a step in common.

But (*) is exactly what is expressed by the consistency criteria [Fair_Module].

4. Follows from 3 by applying 5.8.7(1).

■

5.9 Layered Systems

In this section, systems are introduced as a means to combine modules in a LEGO-like fashion. This aims at building concrete systems using predefined modules from some library. On the system level, these modules are instantiated appropriately and properties for the concrete system (or specification) under consideration may be given using the linear temporal logic presented in Chapter 3.

In most application areas (telecommunications, control, client server) that have been investigated by the TLT group, systems were typically build up in layers. For example, the (software) architecture of a typical control system consists of three layers: On top, there is the so-called process control system (PCS). It consists nowadays of some PC's and is responsible for monitoring and control, for keeping archives, as well as for production planning using, for example, recipes. In the middle, a series of PLCs (for programmable logic controllers) does the intrinsic

[4] Strictly speaking, in order to reduce this theorem to Theorem 4.2.17, it has to be assumed that $\mathcal{F}^n$ only contains weak fairness conditions or all strong fairness conditions can be implemented by weak fairness conditions. This difficulty can be circumvented by doing a direct proof similar to the one presented for the composition theorem on the system level.

control. They poll the physical devices in a cyclic manner in intervals of few milliseconds (compared to reaction times in the range of seconds for the PCS). The bottom layer, finally, sums up the intelligent field devices that come along with their own low level control software (see Figure 5.9).

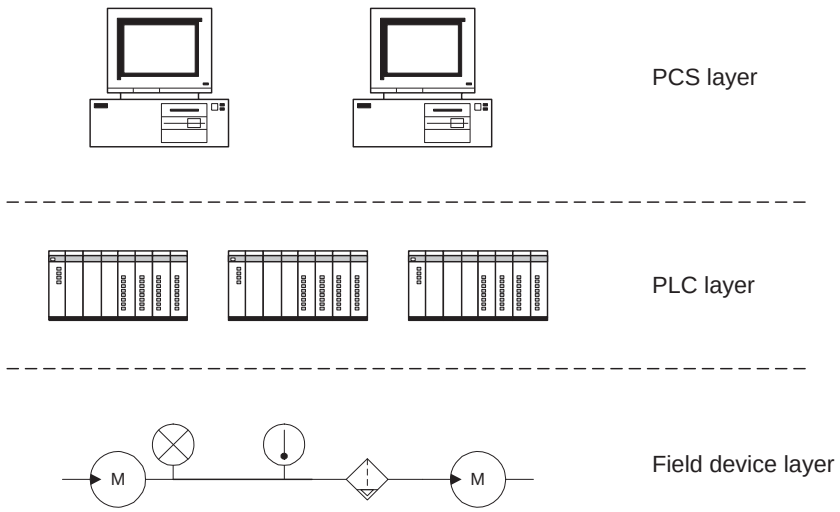


Figure 5.9: The architecture of a typical control system. It is simplified by omitting any concrete bus systems (typically, both TCP/IP and some field bus are used).

A more detailed view on control systems is taken in Chapter 6. At this point, their architecture motivates the introduction of so-called layered systems as an additional means of structuring specifications.[5]

Definition 5.9.1 (Layered Systems)

A tuple $< \mathsf{Sys}, \mathsf{Param}, \mathsf{SysAssumptions}, \mathsf{Layer}_1, \ldots, \mathsf{Layer}_L >$ *is called a system* **iff**

1. Sys *is a name identifying the system,*
2. Param *is a set of parameter declarations* $<$ id, type, expr $>$ *such that no variable occurs in* expr*,*
3. SysAssumptions *is a set of temporal logic formulas describing the assumptions made about the environment of the system,*
4. Layer_i *are tuples* $< \mathsf{Modules}_i, \mathsf{Poperties}_i >$ *where*
   - $\mathsf{Modules}_i$ *is a (finite) set of modules with each module* Mod *being enriched with a (possibly empty) list of substitutions such that*
     [Subst1] *all parameters of* Mod *get replaced by expressions containing only constants or system parameters of appropriate type, and*
     [Subst2] *any identifier occurring in* Mod *may be renamed by a fresh identifier.*
     These substitutions are carried out simultaneously in a preprocessing step, that is, before determing Decls *or extracting the transition relations of the modules.*

[5] Of course, it is possible to use only one layer.

Let Decls *denote the union of all declarations made in any module after all substitutions have been carried out.* Vars *and* Acts *then denote the sets of all declared variables and actions respectively.*

- $\mathsf{Poperties_i}$ *is a set of (arbitrary) temporal formulas, expressing the properties layer* i *guarantees (to layer* $\mathsf{i}+1$*). Properties of the top layer* L *are called the* system properties.

Furthermore, wlog. the set of all modules of all layers is given by $\{\mathsf{Mod_1}, \ldots, \mathsf{Mod_M}\}$.

5. *Any variable (action) is declared at most once in the system as* Write *(*Out*).*

6. *The safety assumptions are fulfilled:*

   For any module Mod, *let* $\Delta^{-\mathsf{Assume}}(\mathsf{Mod}) \stackrel{def}{=} \exists_{\mathsf{hVars,hActs}} \bigwedge_{0 \le \mathsf{i} \le |\mathsf{Blocks(Mod)}|} \Delta(\mathsf{Bl_i^{-Assume}})$

   where $\Delta(\mathsf{Bl^{-Assume}})$ *is obtained from* $\Delta(\mathsf{Bl})$ *by replacing*

   $\forall_{\mathsf{s}}\ (\ \mathsf{in\_ev} \Rightarrow \mathsf{delay} \wedge \mathsf{assume} \wedge \mathrm{Trl}(\mathsf{cmd})\ )$

   in the definition of $\delta_{\mathsf{env}}(\mathsf{Bl})$ *by*

   $\forall_{\mathsf{s}}\ (\ \mathsf{in\_ev} \Rightarrow \mathsf{delay} \wedge \mathrm{Trl}(\mathsf{cmd})\ )$

   for all $<$ tt, s, assume, delay, in_ev, cmd $>$ *occurring in* Bl.

   Further, for any $<$ tt, s, assume, delay, in_ev, cmd $>$ *occurring in one of the modules* Mod, *let* Rely(Mod) *denote the conjunction of all properties below module* Mod *together with the system assumptions.*

   Then, one has to show that

   [Assumptions] $\quad \models \mathsf{Rely(Mod)} \wedge \bigwedge_{1 \le \mathsf{i} \le \mathsf{M}} \Delta^{-\mathsf{Assume}}(\mathsf{Mod_i}) \ \Rightarrow\ \Box\, \forall_{\mathsf{s}}\ (\mathsf{in\_ev} \Rightarrow \mathsf{assume})$

   holds for all $<$ tt, s, assume, delay, in_ev, cmd $>$ *that do not occur in an interfaces.*

7. *Let* Φ^{l} *denote the conjunction of all* $\Phi^{\mathsf{vis}}(\mathsf{Mod})$ *of modules of layer* l. *Similarly,* $\mathsf{P^l}$ *denotes the conjunction of all formulas in* $\mathsf{Properties_l}$ *respectively. Then the following condition has to be satisfied for all layers* l*:*

   [Properties] $\quad \models \mathsf{SysAssumptions} \wedge (\bigwedge_{1 \le \mathsf{k} < \mathsf{l}} \mathsf{P^k}) \wedge \Phi^{\mathsf{l}} \ \Rightarrow\ \mathsf{P^l} \quad$ *for all* $1 \le \mathsf{l} \le \mathsf{L}$

8. *at most one module is delayed by each transition:*

   If delay *differs from* true *for some triggered transition* $<$ tt, s, assume, delay, in_ev, cmd $>$ *then*

   [Delayed_Module] $\quad \mathrm{FAct}(\mathsf{in\_ev}) \subseteq \mathrm{Ctr}(\mathsf{Mod_j})$

   for one module $\mathsf{Mod_j}$.

9. *all visible fairness conditions are executable: Let* $<$ id, F, fg, fcmd $>$ *be some arbitrary visible fairness condition occurring in one of the modules* $\mathsf{Mod_f}$. *Then*

   [Vis_Fair_System]

   $$[\![\, \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta(\mathsf{Mod_f}) \wedge \bigwedge_{\mathsf{tt} \in \mathsf{TriggeredTTs}} (\mathsf{delay} \wedge \mathsf{assume}) \Rightarrow \exists_{\mathsf{vActs(Sys)},\mathsf{vVars'(Sys)}} (\, \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \bigwedge_{1 \le \mathsf{j} \le \mathsf{M}} \delta^{\mathsf{vis}}(\mathsf{Mod_j})\,)\,]\!]$$

$$\textit{for} \quad \mathsf{TriggeredTTs} \stackrel{def}{=} \{\mathsf{tt} \mid < \mathsf{tt}, \mathsf{s}, \mathsf{assume}, \mathsf{delay}, \mathsf{in\_ev}, \mathsf{cmd} > \mathsf{Transitions(Sys)} \;\; \textit{and} \;\; [\![\, \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta(\mathsf{Mod_f}) \;\Rightarrow\; \exists_{\mathsf{s}}\, \mathsf{in\_ev} \,]\!] \}$$

REMARK 1. For assumptions occurring in interfaces, the condition [Assumptions] does not need to be proven because the condition already is checked as a commitment in the inverted module.

Notation:

System Sys

Parameters

param : sort(param) = expr

Layer L:
Properties
[] Prop

Include Module $\mathsf{Mod}_{[\mathsf{id}_1 \leftarrow \mathsf{t}_1, \ldots, \mathsf{id}_k \leftarrow \mathsf{t}_k]}$

⋮

Layer 1:
Properties
[] Prop

Include Module $\mathsf{Mod}_{[\mathsf{id}_1 \leftarrow \mathsf{t}_1, \ldots, \mathsf{id}_l \leftarrow \mathsf{t}_l]}$

Systemassumptions
[] Env_Prop
End. (* System *)

Example: Peterson (continued)

Using the modules Client and Token introduced on Page 128, a system of two modules competing for some mutual exclusive critical section may be build up:

System Peterson

Layer 1:

Properties

$\|$ $\quad \Box \ \neg (\text{cs\_0} \wedge \text{cs\_1})$

$\|$ $\quad (\text{req\_0} \mapsto \text{cs\_0})$

$\|$ $\quad (\text{req\_1} \mapsto \text{cs\_1})$

Include Client $[\text{myToken} \leftarrow 0]$

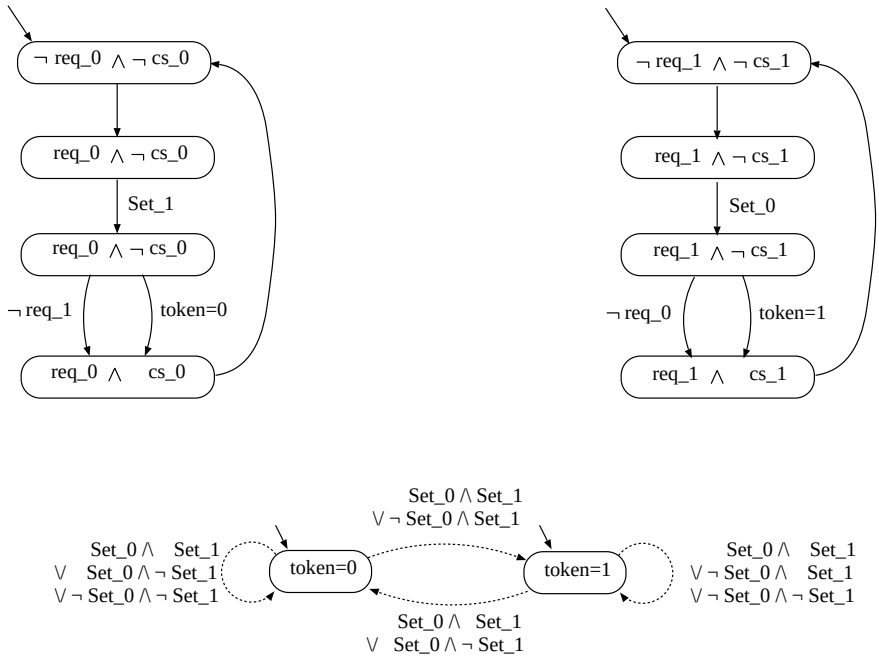
Include Client $[\text{myToken} \leftarrow 1 \text{ , } \text{req\_0} \leftarrow \text{req\_1}, \text{req\_1} \leftarrow \text{req\_0}, \text{cs\_0} \leftarrow \text{cs\_1}, \text{Set\_0} \leftarrow \text{Set\_1}]$

Include Token

End.

Strictly speaking, the two liveness properties could be strengthened to $\Box\Diamond$cs_0 and $\Box\Diamond$cs_1. This is due to local progress which prevents the clients from staying permanently in their idling states.

The following picture shows the BTS corresponding to the instantiated modules occurring in system Peterson.



□

Definition 5.9.2 (Translation to Logic and concrete BTS)

Assuming the notation of Definition 5.9.1,

$$\delta(\mathsf{Sys}) \stackrel{def}{=} \bigwedge_{1 \le i \le M} \delta^{vis}(\mathsf{Mod_i})$$

$$\Delta(\mathsf{Sys}) \stackrel{def}{=} \bigwedge_{1 \le i \le M} \Delta^{vis}(\mathsf{Mod_i})$$

$$\Phi(\mathsf{Sys}) \stackrel{def}{=} \bigwedge_{1 \le i \le M} \Phi^{vis}(\mathsf{Mod_i})$$

$$\Sigma(\mathsf{Sys}) \stackrel{def}{=} \|_{1 \le i \le M} \Sigma^{vis}(\mathsf{Mod_i})$$

where the definition of $\Sigma(\mathsf{Sys})$ *is based on canonical embeddings of the Boolean algebras underlying* $\Sigma^{vis}(\mathsf{Mod_i})$.

By applying [Properties] inductively for all layers, it follows immediately that all properties hold in the system given the environment fulfills the system assumptions:

Corollary 5.9.3

$$\models \quad \mathsf{SysAssumptions} \wedge \Phi(\mathsf{Sys}) \quad \Rightarrow \quad \bigwedge_{\mathsf{P\ in\ Properties(Sys)}} \mathsf{P}$$

Next, two lemmas and a corollary are stated. They will be used in the proof of the composition theorem for systems. The first lemma allows to discard the assumptions:

Lemma 5.9.4

$$\models \quad \mathsf{SysAssumptions} \quad \Rightarrow \quad (\ \Phi^{-\mathsf{Assume}}(\mathsf{Sys}) \quad \Leftrightarrow \quad \Phi(\mathsf{Sys}) \ )$$

Proof:

1. By applying [Properties] inductively for all layers.
2. $\models \quad \mathsf{SysAssumptions} \ \wedge \ \Phi(\mathsf{Sys}) \quad \Rightarrow \quad \Phi^{-\mathsf{Assume}}(\mathsf{Sys})$ holds because by definition,

   $$\models \quad \Phi^{-\mathsf{Assume}}(\mathsf{Sys}) \ \wedge \bigwedge_{\substack{< \mathsf{tt,\ s,\ assume,\ delay,\ in\_ev,\ cmd} > \\ \mathsf{in\ Transitions(Sys)}}} \Box \, \forall_{\mathsf{s}} \, (\mathsf{in\_ev} \Rightarrow \mathsf{assume}) \quad \Leftrightarrow \quad \Phi(\mathsf{Sys}) \quad .$$

   It remains to show

   $$\models \quad \mathsf{SysAssumptions} \ \wedge \ \Phi^{-\mathsf{Assume}}(\mathsf{Sys}) \quad \Rightarrow \quad \Phi(\mathsf{Sys}) \quad .$$

   This is done bottom up for each layer separately. Therefore, let Φ^l denote the conjunction of all $\Phi^{\mathsf{vis}}(\mathsf{Mod})$ of modules of layer l and let P^{l} denotes the conjunction of all formulas in $\mathsf{Properties_l}$.

   Firstly, from [Assumptions], it follows that

   $$\models \quad \mathsf{SysAssumptions} \ \wedge \bigwedge_{1 \le \mathsf{k} < \mathsf{l}} \mathsf{P^k} \ \wedge \ \Phi^{-\mathsf{Assume}}(\mathsf{Sys}) \quad \Rightarrow \bigwedge_{\substack{< \mathsf{tt,\ s,\ assume,\ delay,\ in\_ev,\ cmd} > \\ \mathsf{in\ Transitions(layer\ l)}}} \Box \, \forall_{\mathsf{s}} \, (\mathsf{in\_ev} \Rightarrow \mathsf{assume})$$

   and thus

$$\models \quad \mathsf{SysAssumptions} \wedge \bigwedge_{1 \le k < l} \mathsf{P}^k \wedge \Phi^{-\mathsf{Assume}}(\mathsf{Sys}) \quad \Rightarrow \quad \Phi^l \quad .$$

Using [Properties],

$$\models \quad \mathsf{SysAssumptions} \wedge \bigwedge_{1 \le k < l} \mathsf{P}^k \wedge \Phi^{-\mathsf{Assume}}(\mathsf{Sys}) \quad \Rightarrow \quad \mathsf{P}^l$$

can be deduced which allows to repeat the argument for the next layer. After all, one has shown

$$\models \quad \mathsf{SysAssumptions} \wedge \Phi^{-\mathsf{Assume}}(\mathsf{Sys}) \quad \Rightarrow \bigwedge_{\substack{< \mathsf{tt},\, \mathsf{s},\, \mathsf{assume},\, \mathsf{delay},\, \mathsf{in\_ev},\, \mathsf{cmd} > \\ \text{in } \mathsf{Transitions(Sys)}}} \Box\, \forall_{\mathsf{s}}\, (\mathsf{in\_ev} \Rightarrow \mathsf{assume})$$

as required.

■

Moving from blocks to modules, one had to embed the declarations of the blocks into the declarations of the module. Now embedding is applied to move from modules to systems. Repeating the arguments from Section 5.8, one proves

Lemma 5.9.5 (Embedding)
Let φ_i denote the canonical embedding of the declarations of module $\mathsf{Mod_i}$ *into the declarations of the system* Sys. *Then*

$$\mathbf{Traces}_{\mathsf{Vars(Mod_i)},\mathsf{Acts(Mod_i)}}(\Sigma(\mathsf{Mod_i})) = \mathbf{Traces}_{\mathsf{Vars(Mod_i)},\mathsf{Acts(Mod_i)}}(\Phi(\mathsf{Mod_i}))$$

implies $\mathbf{Traces}_{\mathsf{Vars},\mathsf{Acts}}(\varphi_i\Sigma(\mathsf{Mod_i})) = \mathbf{Traces}_{\mathsf{Vars},\mathsf{Acts}}(\Phi(\mathsf{Mod_i}))$.

as well as

Corollary 5.9.6

$$\bigcap_{1 \le i \le M} \mathbf{Traces}_{\mathsf{Vars},\mathsf{Acts}}(\varphi_i\Sigma(\mathsf{Mod_i})) = \bigcap_{1 \le i \le M} \mathbf{Traces}_{\mathsf{Vars},\mathsf{Acts}}(\Phi(\mathsf{Mod_i}))$$

Finally, in the composition theorem, it is shown
(1) that both $\Phi(\mathsf{Sys})$ and $\Sigma(\mathsf{Sys})$ yield the same set of traces, and
(2) that there is at least one trace.

Theorem 5.9.7

1. $\mathbf{Traces}(\mathsf{Sys}) \stackrel{def}{=} \mathbf{Traces}(\Sigma(\mathsf{Sys})) = \mathbf{Traces}(\Phi(\mathsf{Sys}))$
2. *Assuming the system assumptions,* $\mathbf{Traces}(\mathsf{Sys}) \neq \emptyset$

Proof:

1. $$\mathbf{Traces}(\|_{1 \le i \le M} \Sigma(\mathsf{Mod_i})) \stackrel{Theorem 4.2.10}{=} \bigcap_{1 \le i \le M} \mathbf{Traces}(\varphi_i\Sigma(\mathsf{Mod_i})) \stackrel{Corollary 5.9.6}{=} \bigcap_{1 \le i \le M} \mathbf{Traces}(\Phi(\mathsf{Mod_i})) \stackrel{Lemma 3.2.6}{=} \mathbf{Traces}(\bigwedge_{1 \le i \le M} \Phi(\mathsf{Mod_i}))$$

2. Firstly, assuming the system assumptions, it was shown in Lemma 5.9.4 that
   $\mathbf{Traces}(\Phi(\mathsf{Sys})) = \mathbf{Traces}(\Phi^{-\mathsf{Assume}}(\mathsf{Sys}))$.
   Furthermore,
   $\mathbf{Traces}(\Phi(\mathsf{Mod})) = \mathbf{Traces}(\Phi^{-\mathsf{Commit}}(\mathsf{Mod}))$
   for all modules. Therefore, replacing all assumptions and commitments by true does not influence the set of traces.

   Now, a trace can be defined inductively.

   Firstly, let $\mathsf{VisInitStates}(\mathsf{Mod_m}) \stackrel{\text{def}}{=} \{\xi_0^{\mathsf{m}} \mid (\xi_0^{\mathsf{m}}, \alpha_0^{\mathsf{m}}, \xi_1^{\mathsf{m}}, \dots) \in \mathbf{Traces}(\Phi^{\mathsf{vis}}(\mathsf{Mod_m}))\}$.

   Then $\bigcap_{1 \le \mathsf{m} \le \mathsf{M}} \mathsf{VisInitStates}(\mathsf{Mod_m}) \neq \emptyset$ because each module only constrains its Write-variables and any visible variable may at most once be declared as Write within the system. Thus there is some initial state ξ_0 for the whole system.

   Now assume, an initial trace $(\xi_0, \alpha_0, \xi_1, \dots \xi_{\mathsf{i}})$ is already determined. Then analog to the proof following Theorem 4.2.11, a fairness condition $< \mathsf{fair}, \mathsf{F}, \mathsf{fg}, \mathsf{fcmd} >$ is chosen that "waits" the longest. Let fair occur in Module $\mathsf{Mod_f}$. Now two cases may arise:

   - If the fairness condition is local, then by definition
     $$[\![\mathsf{fg} \;\Rightarrow\; \exists_{\mathrm{Ctr}'(\mathsf{Mod_f})} (\mathrm{Trl}(\mathsf{fcmd}) \wedge \delta(\mathsf{Mod_f}) \wedge \bigwedge_{\mathsf{a} \in \mathsf{vVars}(\mathsf{Mod_f}) \cup \mathsf{vActs}(\mathsf{Mod_f})} \mathsf{Stutter(a)})]\!].$$
     Thus, $\mathsf{Mod_f}$ can do a fair step while stuttering on all visible variables and actions. All other modules simply do a stuttering step.

   - If the fairness condition is visible, then due to [Fair_Module]
     $$[\![\mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta(\mathsf{Mod_f})]\!]^{\mu}$$
     for some valuation μ of the variables and actions declared in $\mathsf{Mod_f}$. Now for all other modules $\mathsf{Mod_j}$ the set of enabled transitions is determined. If there is more than one enabled transition in $\mathsf{Mod_j}$ then, because of [TT_Consistency] and [TT_Consistency_Mod], there is one delay delay implying all others. If this delay is not yet satisfied then module $\mathsf{Mod_j}$ continues doing local steps (that is, with regard to the visible variables and actions the module stutters) until delay holds ([Delay1] and [Delay2]). During that time, all other modules (including $\mathsf{Mod_f}$) do (with regard to the visible variables and actions) stutter steps. After all modules have reached a state where they can accept the fairness step, [Vis_Fair_System] can be applied, and thus
     $$[\![\mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \bigwedge_{1 \le \mathsf{j} \le \mathsf{M}} \delta^{\mathsf{vis}}(\mathsf{Mod_j})]\!]^{\eta_{\mathsf{vis}}}$$
     for some valuation η_{vis} of the visible variables and actions declared in the system. Since $\delta^{\mathsf{vis}}(\mathsf{Mod_j}) \stackrel{\text{def}}{=} \exists_{\mathsf{hVars}(\mathsf{Mod_j}),\mathsf{hActs}(\mathsf{Mod_j}),\mathsf{hVars}'(\mathsf{Mod_j})} \delta(\mathsf{Mod_j})$, it follows further that
     $$[\![\mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \bigwedge_{1 \le \mathsf{j} \le \mathsf{M}} \delta(\mathsf{Mod_j})]\!]^{\eta}$$
     for some valuation η. That is, there is a common step of all modules that fulfills the fairness condition fair.

■

5.10 Parameterized Specifications

In this section, the programming notation will be extended to deal with parameterized specifications. In the context of TLT, parameters may be viewed as an elegant way of writing "generic" specifications. For example, in a paper machine there are groups of drives linked together by a sieve. One drive behaves as the master drive whereas all other drives are slave drives, being controlled by the master drive. However, the number of slave drives varies in different parts of the paper machine as well as in different paper machines. In TLT, such a system could be noted as follows:

System Papermachine

Layer 2:

Include Module Supervisory_Control

End

Layer 1:

Include Module Master_Drive1
Include Module $\mathsf{Slave\_Drives}_{[\mathsf{Number\_of\_slaves} \leftarrow 4]}$

$\vdots$

End

End

In this case, a generic module Slave_Drives is used that can deal with any number of drives. In the concrete system, the parameter Number_of_slaves gets instantiated with 4. Alternatively, it is also possible to replicate a module Slave_Drive four times:

System Papermachine

Layer 2:

Include Module Supervisory_Control

End

Layer 1:

Include Module Master_Drive2
Include $\|_{\langle \mathsf{slave}:[1..4] \rangle}$ Module $\mathsf{Slave\_Drive}_{[\mathsf{x} \leftarrow \mathsf{x}[\mathsf{slave}], \ldots]}$

$\vdots$

End

End

In this case, each of the modules controls exactly one drive. In order to keep these module separate, it is necessary to rename all visible variables and actions. In the example, a visible variable x of module Slave_Drive gets replaced by variables x[1], x[2], x[3], and x[4] respectively.

On the system level at latest, all parameters have to be instantiated. Therefore, although specifications are described parameterized and thus generic, it is always possible to resolve the parameterization syntactically in a preprocessing step. This simplifies the task of implementing, simulating and verifying systems.

Definition 5.10.1 (Vector Types)

A type T *is called vector type* **iff**

(1) $|\mathcal{I}(\mathsf{T})|$ *is finite,*
(2) there is a set C *of constants of type* T *such that* $\mathcal{I} : \mathsf{C} \hookrightarrow\!\!\!\!\rightarrow \mathcal{I}(\mathsf{T})$*, and*
(3) the elements of C *are ordered.*

For example, $\{\mathsf{stopped}, \mathsf{running}, \mathsf{defect}\}$ trivially is a vector type with three constants being ordered implicitly according to their occurrence within the set. $[1..4]$ is a vector type with $\mathsf{C} = \{1, 2, 3, 4\}$. As a special case, vector types may be empty as in $[1..0]$ or $\mathsf{C} = \{\,\}$. This can be useful to describe optional parts of a system.

A first schema is used to declare *vectors* of variables and actions. Consider for example a system with 3 (N) instances (or replicas) of a "generic" pump controller and a supervisory controller. Each pump controller is responsible for supervising one pump (and independent from the other controllers) and uses a Boolean valued variable $\mathsf{pump\_ok}[\mathsf{i}]$ to denote the status of its (that is the i-th) pump. For the supervisory controller however it might be reasonable to write an interface dealing with all 3 (N) pump controllers. Thus, from the view of this interface, there exists a Boolean valued vector $(\mathsf{pump\_ok}[1], \mathsf{pump\_ok}[2], \mathsf{pump\_ok}[3])$ of length 3 (N). Declarations for vectors of length N are written

$$\mathrm{class}(\mathsf{a}) \quad \mathsf{a} \quad : \quad \mathsf{Vector}\ [1..\mathsf{N}]\ \mathsf{Of}\ \mathrm{sort}(\mathsf{a})$$

which is used as schema to declare the following N variables or actions:

$$\mathrm{class}(\mathsf{a}[1]) \quad \mathsf{a}[1], \ldots, \mathsf{a}[\mathsf{N}] \quad : \quad \mathrm{sort}(\mathsf{a}) \quad .$$

More generally, vectors may be of arbitrary vector types:

$$\mathrm{class}(\mathsf{a}) \quad \mathsf{a} \quad : \quad \mathsf{Vector}\ \mathsf{VectorType}\ \mathsf{Of}\ \mathrm{sort}(\mathsf{a}) \quad .$$

If $\mathsf{VectorType}$ is determined by N constants $\{\mathsf{c}_1, \ldots, \mathsf{c}_\mathsf{N}\}$, then the N variables or actions declared implicitly by this construction are

$$\mathrm{class}(\mathsf{a}[\mathsf{c}_1]) \quad \mathsf{a}[\mathsf{c}_1], \ldots, \mathsf{a}[\mathsf{c}_\mathsf{N}] \quad : \quad \mathrm{sort}(\mathsf{a}) \quad .$$

It is important to realize that by this construction N variables or actions get declared. This distinguishes vectors from arrays: For an array a, $\mathsf{a}(1)$ and $\mathsf{a}(2)$ refer to two positions of one array variable, whereas $\mathsf{a}[1]$ and $\mathsf{a}[2]$ refer to two different variables or actions.

In the sequel, let VT denote a vector type that is determined by N constants $\{\mathsf{c}_1, \ldots, \mathsf{c}_\mathsf{N}\}$. Then the following schemata are introduced to handle vectors:

1. For formulas, $\bigwedge_{\langle \mathsf{c}:\mathsf{VT}\rangle} \mathsf{p}$ abbreviates the conjunction $\mathsf{p}_{[\mathsf{c}\leftarrow\mathsf{c}_1]} \wedge \ldots \wedge \mathsf{p}_{[\mathsf{c}\leftarrow\mathsf{c}_\mathsf{N}]}$.

   For example, let x be a variable of type $\mathsf{Vector}\ [1..3]\ \mathsf{Of}\ \mathsf{Integer}$.
   Then $\bigwedge_{\langle \mathsf{c}:[1..3]\rangle} \mathsf{x}[\mathsf{c}] = 0$ abbreviates $\mathsf{x}[1] = 0 \wedge \mathsf{x}[2] = 0 \wedge \mathsf{x}[3] = 0$.

   Disjunctions are handled analog.

2. On command level, $\langle c : VT\rangle$ cmd abbreviates $\mathsf{cmd}_{[c\leftarrow c_1]} \parallel \cdots \parallel \mathsf{cmd}_{[c\leftarrow c_n]}$.

3. For triggered transitions,
   $\langle$ c : VT $\rangle$ $[\mathsf{tt}]_s$ {assume} $\langle$delay$\rangle$ ev $=\triangleright$cmd
   abbreviates N triggered transitions

   $[\mathsf{tt\_c}_1]$ $\{\mathsf{assume}_{[c\leftarrow c_1]}\}$ $\langle\mathsf{delay}_{[c\leftarrow c_1]}\rangle$ $\mathsf{ev}_{[c\leftarrow c_1]}$ $=\triangleright\mathsf{cmd}_{[c\leftarrow c_1]}$
   $\vdots$
   $[\mathsf{tt\_c}_n]$ $\{\mathsf{assume}_{[c\leftarrow c_n]}\}$ $\langle\mathsf{delay}_{[c\leftarrow c_n]}\rangle$ $\mathsf{ev}_{[c\leftarrow c_n]}$ $=\triangleright\mathsf{cmd}_{[c\leftarrow c_n]}$.

   This notation extends naturally to tuples of specification variables:
   $\langle\ c_1^1, \ldots, c_{n_1}^1 : \mathrm{VT}^1\ ,\ldots,\ c_1^m, \ldots, c_{n_m}^m : \mathrm{VT}^m\ \rangle$ $[\mathsf{tt}]_s$ {assume} $\langle$delay$\rangle$ ev $=\triangleright$cmd
   In this case, $\prod_{1\le i\le m} n_i * N^i$ transitions arise from syntactically substituting the $\sum_{1\le i\le m} n_i$ vector indices $[c_j^i]$ with all possible combinations of constants.

4. Similarly, for guarded transitions, the following schema is introduced:
   $\langle$ c :VT $\rangle$ [gt] guard $\rightarrow$ cmd
   This schema abbreviates N guarded transitions

   $[\mathsf{gt\_c}_1]$ $\mathsf{guard}_{[c\leftarrow c_1]} \rightarrow \mathsf{cmd}_{[c\leftarrow c_1]}$
   $\vdots$
   $[\mathsf{gt\_c}_n]$ $\mathsf{guard}_{[c\leftarrow c_n]} \rightarrow \mathsf{cmd}_{[c\leftarrow c_n]}$.

   Again, this notation extends naturally to tuples of specification variables:
   $\langle\ c_1^1, \ldots, c_{n_1}^1 : \mathrm{VT}^1\ ,\ldots,\ c_1^m, \ldots, c_{n_m}^m : \mathrm{VT}^m\ \rangle$ [gt] guard $\rightarrow$ cmd

5. For fairness conditions,
   $\langle$ c : VT $\rangle$ [fair] F(fg, fcmd)
   abbreviates N fairness conditions:

   $[\mathsf{fair\_c}_1]$ $\mathsf{F}(\mathsf{fg}_{[c\leftarrow c_1]}, \mathsf{fcmd}_{[c\leftarrow c_1]})$
   $\vdots$
   $[\mathsf{fair\_c}_n]$ $\mathsf{F}(\mathsf{fg}_{[c\leftarrow c_n]}, \mathsf{fcmd}_{[c\leftarrow c_n]})$.

   Again, this notation extends naturally to tuples of specification variables:
   $\langle\ c_1^1, \ldots, c_{n_1}^1 : \mathrm{VT}^1\ ,\ldots,\ c_1^m, \ldots, c_{n_m}^m : \mathrm{VT}^m\ \rangle$ [fair] F(fg, fcmd)

6. For blocks,
   $\langle$c : VT$\rangle$ Block $\mathsf{Bl}[\mathsf{id}_1 \leftarrow \mathsf{t}_1, \ldots, \mathsf{id}_m \leftarrow \mathsf{t}_m]$
   abbreviates N blocks

   Block ($\mathsf{Bl\_c}_1[\mathsf{id}_1 \leftarrow \mathsf{t}_1, \ldots, \mathsf{id}_m \leftarrow \mathsf{t}_m]$)$[\mathsf{c} \leftarrow \mathsf{c}_1]$
   $\vdots$
   Block $(\mathsf{Bl\_c}_n[\mathsf{id}_1 \leftarrow \mathsf{t}_1, \ldots, \mathsf{id}_m \leftarrow \mathsf{t}_m])[\mathsf{c} \leftarrow \mathsf{c}_n]$

   such that in a first step all substitutions from $[\mathsf{id}_1 \leftarrow \mathsf{t}_1, \ldots, \mathsf{id}_m \leftarrow \mathsf{t}_m]$ are carried out, and in a second step all identifiers c get replaced by an identifier c_i for each constant c_i of the vector type.

Again, this notation extends naturally to tuples of specification variables:

$\langle\ \mathrm{c}^1_1, \ldots, \mathrm{c}^1_{n_1} : \mathrm{VT}^1\ , \ldots,\ \mathrm{c}^m_1, \ldots, \mathrm{c}^m_{n_m} : \mathrm{VT}^m\ \rangle$ Block BI

7. Within systems, vectors of modules are described exactly the same way as vectors of blocks are described in the context of modules.

Example 1: A token for N modules

The following module generalizes module Token presented on page 128:

Module GeneralToken

Parameters

N : Integer

Declarations

Variables

Write token : [0..N-1]

Actions

In Set : Vector [0..N-1] Of ()

Transitions

$$[\mathrm{set}] \quad \bigvee_{\langle pr:[0..N\text{-}1]\rangle} \mathrm{Set}[\mathrm{pr}] \ \Rightarrow\!\triangleright\ \mathrm{token}' \in \{\ \mathrm{pr} : [0..N\text{-}1]\ \mid\ \mathrm{Set}[\mathrm{pr}]\ \}$$

End.

If N equals two then the sole transition becomes

$$[\mathrm{set}] \quad \mathrm{Set}[0] \vee \mathrm{Set}[1] \ \Rightarrow\!\triangleright\ \mathrm{token}' \in \{\ \mathrm{pr} : [0..1]\ \mid\ \mathrm{Set}[\mathrm{pr}]\ \}$$

which yields the same translation as module Token, namely

$$\begin{aligned} \Phi^{\mathsf{vis}}(\mathsf{GeneralToken}) &= \Phi(\mathsf{GeneralToken}) \\ &= \Box\ (\ \mathrm{token}' \neq \mathrm{token} \Rightarrow \begin{array}{ll} & Set[0] \wedge \neg Set[1] \wedge token' = 0 \\ \vee & \neg Set[0] \wedge Set[1] \wedge token' = 1 \\ \vee & Set[0] \wedge Set[1] \wedge token' = token' \end{array} \ )\ . \end{aligned}$$

□

Example 2: A round robin scheduler

This example presents a complete system made up of a scheduler that is capable of starting N processes in a round robin fashion. Although the example is rather small, it demonstrates nearly all kinds of parameterizations.

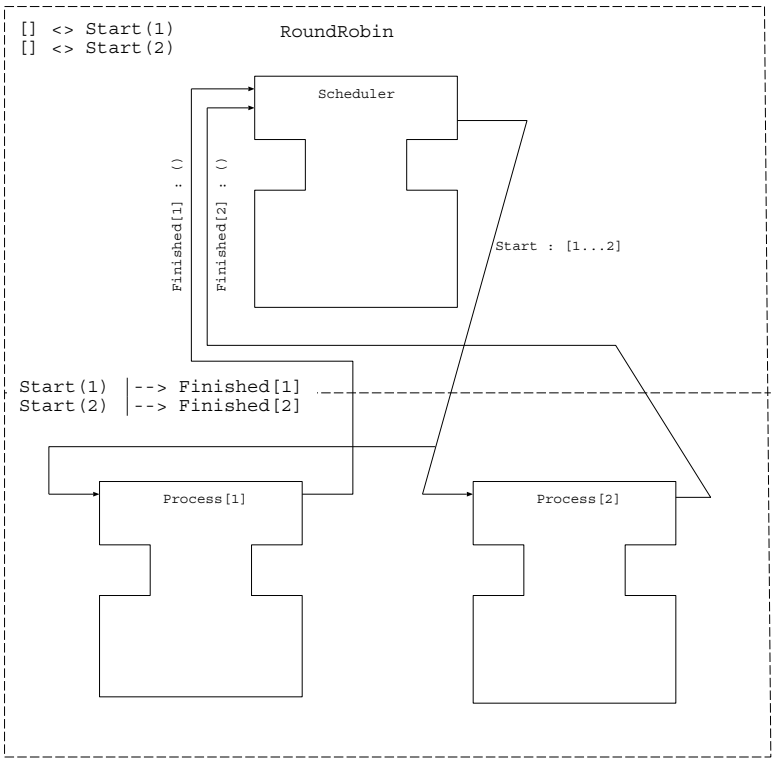


Figure 5.10: The architecture of system RoundRobin.

Module Scheduler

Parameters

N : Natural

Declarations

In	Finished :	Vector [1..N] Of ()
Out	Start	: [1..N]
History	pnd	: [1..N] init 1
Local	start	: Boolean init true

Transitions

	[gt]	start →	¬start′ ‖ Start(pnd)
⟨ pr : [1..N] ⟩	[tt]	{ pnd = pr ∧ ¬start }	Finished[pr] =▷ (If pr<N Then pnd′= pnd+1 Else pnd′= 1) ‖ start′

End

Module Process

Parameters

N : Natural % number of processes in whole system
Me : Natural % identifies this process within system

Declarations

In Start : [1..N]
Out Finished : ()
Local pc : [1..3] init 1

Transitions

$[gt_1]$ { pc=1 } Start(Me) $\Rightarrow\triangleright$ pc′=2
$[gt_2]$ pc=2 $\rightarrow$ pc′=3 || ...
$[gt_3]$ pc=3 $\rightarrow$ pc′=1 || Finished

End

System RoundRobin

Parameters

ProcNo : Natural := 2

Properties

$[\!] \bigwedge_{pr:[1..2]} \Box\Diamond$ Start(pr)

Layer 2:

Include Module Scheduler$[N \leftarrow ProcNo]$

End

Properties

$[\!] \bigwedge_{pr:[1..2]}$ (Start(pr) $\mapsto$ Finished[pr])

Layer 1:

Include $\langle$ pr : [1..ProcNo] $\rangle$ Module Process$[N \leftarrow ProcNo, Me \leftarrow pr , Finished \leftarrow Finished[pr]]$

End

End

5.11 Some Notes Concerning Implementation

All entities of the programming language presented in this chapter have to satisfy both syntactic and semantic consistency criteria. Typically there is some trade off between syntactic and semantic consistency criteria: Whereas syntactic criteria are easy to check, they might be too "coarse" and thus restrict the expressibility of the language too much. More often, there is danger that they lead to specifications that look unnatural or even unreadable. On the other hand, semantic criteria can be adjusted to exactly what is necessary to obtain the desired (e.g. compositionality) results. However, at the prize of getting proof obligations that may be hard to check or even undecidable.

In the sequel, semantical consistency criteria are replaced by syntactical ones and the effect on implementation is discussed.

Transitions

The sole semantic consistency criteria of transitions is the requirement that they are executable. This may be achieved in essence by allowing only assignments instead of arbitrary transition predicates.

Modules

On the level of modules, difficulties concerning implementation may arise due to contradicting transitions that have to be executed simultaneously. Within one block $\mathsf{TT\_Consistency}$ forbids these inconsistencies. A simple purely syntactic substitute is given by

[Syn_TT_Consistency] $\mathrm{Ctr}(cmd_i) \cap \mathrm{Ctr}(cmd_j) = \emptyset$ for all $1 \leq i, j \leq |tt|$, $i \neq j$

The problem of inconsistencies resulting from "cyclic" triggered transitions in different blocks may be tackled by a dependency graph defined as follows:

Definition 5.11.1 (Dependency Graph)

Given a module made up of a set of blocks $\mathsf{Bl} \stackrel{def}{=} \{\mathsf{Bl}^0, \ldots, \mathsf{Bl}^N\}$, *the corresponding dependency graph is defined as a directed graph* $\mathsf{DG(Mod)} = (\mathsf{V}, \mathsf{I}, \mathsf{E})$ *where*

- *the nodes are given as*

  $$\mathsf{V} \stackrel{def}{=} \bigcup_{0 \leq i \leq N} (\{ \{\mathsf{a}\} \mid \mathsf{a} \in \mathrm{Ctr}(\mathsf{Bl}^i) \textit{ and } a \textit{ does not occur in any guarded transition} \} \cup \{ \{\mathsf{a} \in \mathrm{Ctr}(\mathsf{Bl}^i)\} \mid a \textit{ occurs in some guarded transition of block } \mathsf{Bl}^i \}) ,$$

- *the initial nodes are defined as*
  $\mathsf{I} \stackrel{def}{=} \bigcup_{0 \leq i \leq N} \{ \{\mathsf{a} \in \mathrm{Ctr}(\mathsf{Bl}^i)\} \mid a \textit{ occurs in some guarded transition of block } \mathsf{Bl}^i \}$, *and*

- $(\mathsf{v}_1, \mathsf{v}_2) \in \mathsf{E} \subseteq \mathsf{V} \times \mathsf{V}$ **iff** *there is a triggered transition* tt *such that*

  - *some* $\mathsf{a}_1 \in \mathsf{v}_1$ *occurs in the event* ev *of* tt *and*
  - *some* $\mathsf{a}_2 \in \mathsf{v}_2$ *occurs in the command* cmd *of* tt.

DG(Mod) *is said to be weak acyclic* **iff** *no cycle in* DG(Mod) *can be reached starting from a node* $\mathsf{v} \in \mathsf{I}$.

Intuitively, the initial nodes I of a dependency graph mark sets of actions (and variables) that might be forced to happen (to be updated) due to fairness conditions like local progress. The additional nodes in V are made up of singletons of actions and variables that are used only in triggered transitions. Thus, following the edges in the dependency graph, all actions may be determined that might be triggered in a step of the module.

Due to [ExclCtr], the controlled actions of the blocks are disjoint. Due to [Ev], actions do not occur in any event of the block by which they are controlled. These two consistency conditions allow to deduce immediately from the definition of the dependency graph that

Corollary 5.11.2

1. $\mathsf{v}_1 \cap \mathsf{v}_2 = \emptyset$, *and*
2. $(\mathsf{v}, \mathsf{v}) \notin \mathsf{E}$.

Clearly, if a graph DG(Mod) is weak acyclic then (due to [TT]) it is possible to determine valuations for all actions by successively following the edges in the graph (and setting all actions not reached to $\bot$).

To give an example, consider the following three blocks:

```
Block  Block1
  Declarations
    Out     A    : ()
  Transitions
    [gt1]  true  → A
  Fairness
    [wf1]WF(true,A)
End
```

```
Block  Block2
  Declarations
    Out     A,B,C : ()
  Transitions
    [tt2]  A ∧ ¬ B =▷C
End
```

```
Block  Block3
  Declarations
    Out     B,C    : ()
  Transitions
    [tt3]  C =▷B
End
```

In this example, A is the sole action that (always eventually) is forced" by the fairness condition $[\mathsf{wf}_1]$. The dependency graph with regard to this initial state is not weak acyclic:

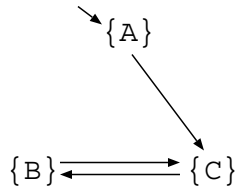


Figure 5.11: The dependency graph with regard to the initial state A.

As may be seen from the dependency graph, the triggered transitions tt_2 and tt_3 depend on each other. Indeed they constitute a contradiction:

(1) Suppose $A \wedge \neg B$ "occurs". Then tt_2 triggers C and tt_3 triggers B, which contradicts $A \wedge \neg B$.

(2) Now suppose $A \wedge B$ "occurs". Then the second block does not trigger C (recall that $\delta(\mathsf{Block2}) \stackrel{\text{def}}{=} C \Leftrightarrow A \wedge \neg B$) and therefore the third block does not trigger B (because $\delta(\mathsf{Block3}) \stackrel{\text{def}}{=} B \Leftrightarrow C$) which contradicts $A \wedge B$.

Together, it follows that A may never happen. However, the fairness condition in the first block forces always eventually A and therefore the composition of the three blocks is not executable.

If no fairness was required for A, then the dependency graph is (by definition) weak acyclic and there is a model for the composition, namely

$$\begin{pmatrix}\alpha_0(A)=\bot\\ \alpha_0(B)=\bot\\ \alpha_0(C)=\bot\end{pmatrix} \longrightarrow \begin{pmatrix}\alpha_1(A)=\bot\\ \alpha_1(B)=\bot\\ \alpha_1(C)=\bot\end{pmatrix} \longrightarrow \ldots \quad .$$

Unfortunately, there are also situations where a dependency graph based only on syntactic informations is too coarse: If in the example tt_2 gets replaced by

$[\mathsf{tt}_{2a}] \quad \mathrm{A} \wedge \mathrm{B} \Rightarrow\!\triangleright \mathrm{C} \quad ,$

then the dependency graph does not change and forbids the composition of the three blocks. Nevertheless there are executions. For example,

$$\begin{pmatrix}\alpha_0(A)=\surd\\ \alpha_0(B)=\surd\\ \alpha_0(C)=\surd\end{pmatrix} \longrightarrow \begin{pmatrix}\alpha_1(A)=\surd\\ \alpha_1(B)=\surd\\ \alpha_1(C)=\surd\end{pmatrix} \longrightarrow \ldots \quad .$$

□

The dependency graph may now be used to replace the semantic criteria [Fair_Module] in the definition of modules:

[Syn_Fair_Module] The dependency graph $\mathsf{DG(Mod)}$ is weak acyclic.

Using this criteria, the proof of fairness independence in Theorem 5.8.8(3) has to be adjusted:

Again, let $v^n \in V^n$ and in addition for some fixed but arbitrary k let $v^k \in G$ for some $\mathsf{WF}(G,E,L) \in \mathcal{F}(\mathsf{Bl}^k)$.
Then, it has to be shown that if for all $n \neq k$ there are $w^n \in V^n$ and $w^k \in V^k$ such that $\emptyset \neq \varphi_k(L(v^k,w^k)) \cap \varphi_n(M^n(v^n,w^n))$ then it holds that

$$\emptyset \neq \varphi_k(L(v^k,\tilde{w}^k)) \cap \bigcap_{0\leq n\leq N} \varphi_n(M^n(v^n,\tilde{w}^n)) \text{ for some } \tilde{w}^n \in V^n.$$

This time, the dependency graph is used in order to successively determine a valuation μ:

1. $\mu(x) \stackrel{\text{def}}{=} v^n(x)$ for all $x \in \mathrm{Ctr}_V(\mathsf{Bl}^n)$
2. $\mu(x) \stackrel{\text{def}}{=} arb.$ for all $x \in \mathsf{Vars} - \mathrm{Ctr}_V(\mathsf{Bl}^n)$
3. choose $\mu(a)$ for all $a \in \mathrm{Ctr}'(\delta_{ctr}(\mathsf{Bl}^k))$ such that $[\![\, \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{Bl}^k) \,]\!]^\mu$
4. $\mu(A) \stackrel{\text{def}}{=} \bot$ for all $A \in \mathsf{Acts} - \bigcup_{0\leq n\leq N} \mathrm{Ctr}_{Act}(\mathsf{Bl}^n)$
5. Now all nodes $v \in V$ of $\mathsf{DG(Mod)}$ without predecessors are determined. Then, for all actions $A \in v$, let $\mu(A) \stackrel{\text{def}}{=} \bot$ and for all variables $x \in v$, let $\mu(x') \stackrel{\text{def}}{=} \mu(x)$.

6. Next, successively all nodes $v \in V$ are determined whose predecessors already are fixed. Then, for all actions $A \in v$ that occur in a transition with $[\![\mathsf{ev}]\!]^\mu$, choose $\mu(A)$ according to [TT], and for all variables $x \in v$ that occur in a transition with $[\![\mathsf{ev}]\!]^\mu$, choose $\mu(x')$ according to [TT]. For all remaining actions $A \in v$, let $\mu(A) \stackrel{\text{def}}{=} \bot$ and for all remaining variables $x \in v$, let $\mu(x') \stackrel{\text{def}}{=} \mu(x)$.

   Because of [TT_Consistency] (no contradictions within one block), [ExclCtr] (no contradictions between triggered transitions of different blocks) and [Syn_Fair_Module] (no action or variable is "visited" twice), this results in a well defined valuation.

7. Finally, $\mu(A) \stackrel{\text{def}}{=} \bot$ for all remaining actions A from $\mathrm{Ctr}_{Act}(\mathsf{mod})$ and $\mu(x') \stackrel{\text{def}}{=} \mu(x)$ for all remaining variables x from $\mathrm{Ctr}_V(\mathsf{mod})$.

Now, μ is completely determined and besides all $\delta_{env}(\mathsf{Bl}^n)$ also all $\delta_{ctr}(\mathsf{Bl}^n)$ are fulfilled. Therefore, as required $\mu \in \varphi_k(L(v^k, \tilde{w}^k)) \cap \bigcap_{0 \le n \le N} \varphi_n(M^n(v^n, \tilde{w}^n))$ holds by choosing $\tilde{w}^n(x) \stackrel{\text{def}}{=} \mu(x)$ for all $x \in \mathrm{Ctr}(\mathsf{Bl}^n)$.

On the system level, finally, there also exists the problem of inconsistencies resulting from "cyclic" triggered transitions. Again, this may be tackled by a dependency graph defined as follows:

Definition 5.11.3 (System Dependency Graph)

Given a system made up of a set of modules $\mathsf{Sys} \stackrel{def}{=} \{\mathsf{Mod}_1, \ldots, \mathsf{Mod}_\mathsf{M}\}$, *the corresponding dependency graph is defined as a directed graph* $\mathsf{DG(Sys)} = (\mathsf{V}, \mathsf{I}, \mathsf{E})$ *where*

- *the nodes are given as*

  $$\mathsf{V} \stackrel{def}{=} \bigcup_{1 \le \mathsf{i} \le \mathsf{M}} (\{ \{\mathsf{a}\} \mid \mathsf{a} \in \mathrm{Ctr}(\mathsf{Mod_i}) \text{ and } a \text{ is visible and } a \text{ does not occur in any guarded transition} \} \cup \{ \{\mathsf{a} \in \mathrm{Ctr}(\mathsf{Mod_i})\} \mid a \text{ is visible and } a \text{ occurs in some guarded transition of } \mathrm{Ctr}(\mathsf{Mod_i}) \}) ,$$

- *the initial nodes are defined as*

  $$\mathsf{I} \stackrel{def}{=} \bigcup_{1 \le \mathsf{i} \le \mathsf{M}} \{ \{\mathsf{a} \in \mathrm{Ctr}(\mathsf{Mod_i})\} \mid a \text{ is visible and } a \text{ occurs in some guarded transition of } \mathrm{Ctr}(\mathsf{Mod_i}) \}) , \text{ and}$$

- $(\mathsf{v}_1, \mathsf{v}_2) \in \mathsf{E} \subseteq \mathsf{V} \times \mathsf{V}$ **iff** *there is a triggered transition* tt *such that*
  - *some* $\mathsf{a}_1 \in \mathsf{v}_1$ *occurs in the event* ev *of* tt *and*
  - *some* $\mathsf{a}_2 \in \mathsf{v}_2$ *occurs in the command* cmd *of* tt.

$\mathsf{DG(Sys)}$ *is said to be weak acyclic* **iff** *no cycle in* $\mathsf{DG(Sys)}$ *can be reached starting from a node* $\mathsf{v} \in \mathsf{I}$.

Chapter 6

The Fault Tolerant Production Cell

6.1 Introduction

The case study "Fault Tolerant Production Cell" was defined by the FZI (Forschungszentrum Informatik) at Karlsruhe for the KorSys (correct software for safety critical systems) project. Besides the FZI, also the University of Oldenburg, the Technical University of Munich, BWM, ESG, and Siemens participate in KorSys. The case study is an extension of the case study "Production Cell" introduced in [LL95], that has been worked out using more than 20 different approaches (among these is also a TLT solution, see [CH95]).

The "Fault Tolerant Production Cell" extends the original task description mainly due to the fact that all devices may fail. To cope with these failures, additional sensors have been added.

The overall architecture of the production cell is shown in Figure 6.1.

The production cycle for a single metal plate does not differ from the process in [LL95]. If the signal at the beginning of the feed belt shows green, a plate may be added to the feed belt. The plate is conveyed to the elevating rotary table. The robot picks the plate up using its first arm, and puts the plate into a unoccupied press. Here, the piece is processed and afterwards picked up with the second arm of the robot. The plate is carried to the deposit belt. If the traffic light at the end of the deposit belt shows green, the plate may be forwarded to some undefined environment.

The production cell is an example for a reactive control (also called embedded) system, that is, a system consisting of (a set of) physical devices and (a set of) controlling devices that interact by means of sensors and actuators (see Figure 6.2)

A model for an embedded system thus consists of

1. explicitly modeled physical devices PD_i,
2. explicitly modeled controllers CD_i, and
3. the environment that is modeled implicitly in form of assumptions .

Explicit modeling of physical devices allows expressing "mixed" properties like "if in the real world no component gets broken, then the controllers will report no errors". If the model would

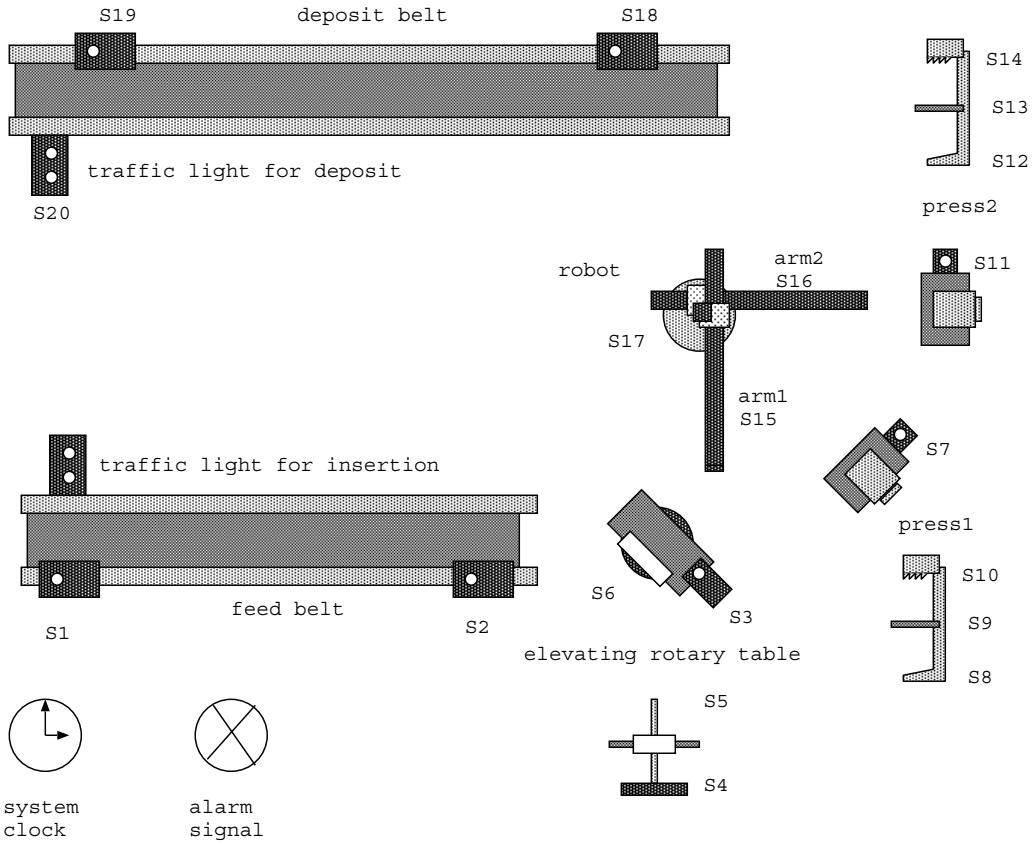


Figure 6.1: The architecture of the fault tolerant production cell (top view and side view).

"end" with the observation of the sensors, one could only "indirectly" argue about the physical devices, like "if the sensors show behavior B then the controllers will report no errors". An engineer formulating such a property would of course implicitly (using his knowledge about the physical devices) identify the observed behavior B with the situation "no error occurs", however this step is a frequent source of errors and it is clearer to explicitly model the physical devices.

With both the physical devices and their controllers being modeled as TLT specifications, the whole production cell may be described as a two-layered system (see Figure 6.3).

According to Figure 5.9, one might expect a third layer with higher level control functionality like diagnosis or production planning. However, because almost all errors can be dealt with locally, a general diagnosis module is not necessary for the case study. And as far as production planning is concerned, it also can be dealt with locally with the sole exception that the robot needs knowledge about the state of the presses to decide to which one the next plate has to be passed. Because an extra module would have increased the necessary communication[1], the "planning" was moved to the controller for the robot.

[1]With regard to the distributed implementation on a LAN (see [CHB96b]), the communication turns out to be the main obstacle for efficiency.

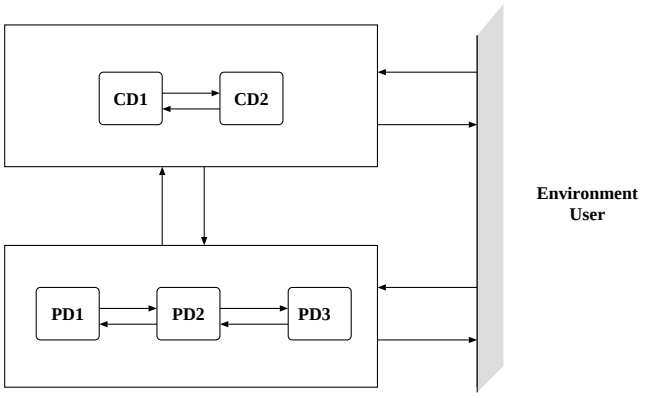


Figure 6.2: An embedded system.

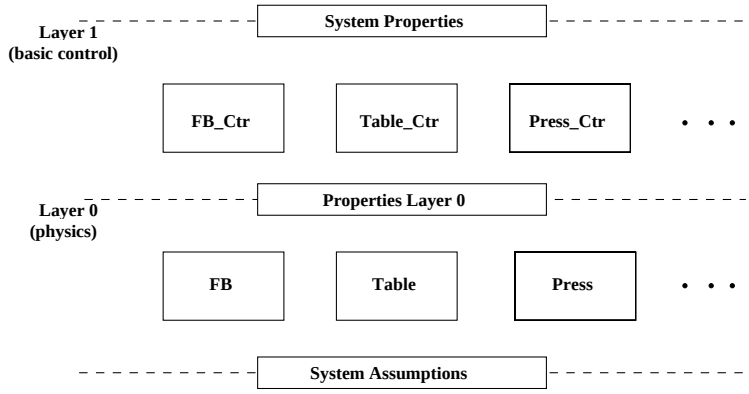


Figure 6.3:

6.2 A Model of the Production Cell

In order to extract a discrete model of some physical system, one typically has to proceed in several stages: The first model might be an continuous model (like a set of differential equations) describing trajectories of the system in time. A second model may introduce discrete points of time where sensors are read and actuators are set. Finally, one might end up with a finite, discrete model described as an automaton. Each "refinement" step has to be validated by relating the models in such a way that it is guaranteed that no (relevant) possible behavior are lost. For the production cell case study, the physics are "defined" by the textual task description. For example, maximal times are assumed without proof in whatever model. As nothing is known about the underlying physics, the specifications presented here already start at the discrete (automaton) level.

The models of physical devices simply describe what can happen. I.e. they only exclude what physically is impossible (like a sensor detecting a plate if there is none). Thus these specifications typically allow many traces and are easily validated with respect to the textual task description.

To describe physical behavior appropriately, TLT specifications are used with only one difference to the TLT specifications introduced so far for modeling software: The fairness section gets replaced by a general (temporal) formulas section allowing to specify arbitrary liveness properties

but also invariants fulfilled by the physical devices. Especially, together with the fairness section, the minimal fairness (local progress) requirement is given up for physical devices. Recall that for a specification to be run on a computer, it is necessary that it is realizable which can not be guaranteed for arbitrary liveness requirements. For physical devices one needs not care about schedulers and implementation, thus this restriction does not make sense.

The hierarchy block - module - system fits very well to model sensors/actors - physical devices - the complete production cell. Thus this hierarchy is kept for the description of the physical system. As an alternative, one might also specify the sensors and actors as modules that might be arranged in a separate layer. Variables and actions now are to be interpreted as an abstract means to describe physical behavior in an operational style. Actions, for example, are used to express that two physical devices do something commonly, in one instant of time.

In this case study, timing conditions are used solely to supervise the movement of devices. For this purpose, it is sufficient to use abstract timer actions (like, for example, StartTimerTable and TimeOutTable). If more complicated real time computations are necessary, real time may be modeled by introducing a real valued variable t representing the current time as in [AL92].

6.2.1 Modeling Sensors and Actuators

The sensors and actuators occurring in the production cell may be divided into 6 types:

1. Boolean sensors,
2. traffic lights,
3. belt motors (moving in one direction),
4. motors moving in two directions (e.g. the vertical table movement),
5. motors supervised by an "intelligent" sensor (e.g. the horizontal table movement), and
6. the alarm signal set by the controllers in case of failure and reset by the environment after repair.

Because all sensors and actors are needed in the sequel, they are now briefly introduced.

Sensors

About Boolean sensors, the task description says:

> "Boolean sensors may fail. In this case, they return a zero value."

Thus three states are to be distinguished : a sensor may detect something, it may work but not detect anything, or it may be defect. From the controllers view, the last two states can not be distinguished. This is reflected by the fact that a controller reading the sensor value, in both states gets 0 as a result.

In the following specification, the variable sensor denotes the actual state the sensor is in, whereas the variable s denotes the value visible to the controllers.

```
Block  Sensor

  Declarations

    Variables

      Write     sensor   : {detecting, not_detecting, defect}
      Write     s        : Boolean

  Transitions

    ‖  true ⟶  sensor′∈ { detecting, not_detecting, defect }  ‖
               If  sensor′= detecting  Then  s′  Else  ¬ s′

End.
```

An immediate consequence from this specification is the property

$\Box$ (sensor = detecting $\Leftrightarrow$ s)

denoting the fact, that the recognition of plates or positions is trustworthy. A visualization of block Sensor is given in Figure 6.4.

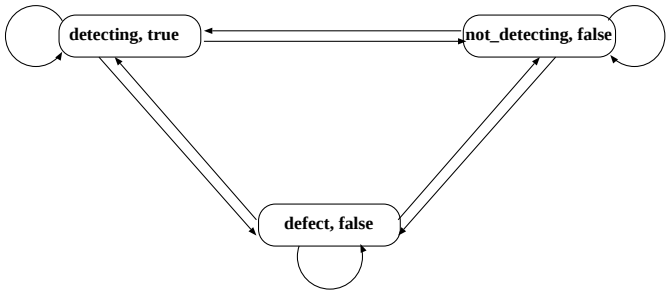


Figure 6.4: BTS corresponding to block Sensor.

The specification for the Boolean sensor is used 15 times within the production cell:

Sensor	**Location**	**Function**
s1 / s2	feed belt	detect plate at beginning / end
s3	table	detect plate
s4 / s5	table	detect position at bottom / top
s7 (s11)	press1 (press2)	detect plate
s8 / s9 / s10 (s12 / s13 / s14)	press1 (press2)	detect position at bottom / middle / top
s18 / s19	deposit belt	detect plate at beginning / end

REMARK 1. According to the command sensor′ ∈ {detecting, not_detecting, defec},
a sensor in general can change its state arbitrarily. This is no longer true when the sensor is placed in some concrete environment, namely as part of a physical device. E.g.

1. The sensor detecting whether the table is in top position can only be in state detecting if the table actually is there:
   $\Box$ (sensor5 = detecting $\Rightarrow$ pos_t_v = at_top)
2. A plausible (but not used) error scenario:
   $\Box\Diamond$ sensor5 $\neq$ defect

REMARK 2. To detect plates, the sensors used in this case study register changes of an electro-magnetic field when plates come close. If they are defect, they report "no plate". Another type of sensors, photo electric cells, report "plate" in case they are defect. Thus this kind of sensor is modeled by replacing

If sensor′= detecting Then s′ Else ¬ s′

by

If sensor′= detecting Then ¬ s′ Else s′.

Combining the two types of sensors, one obtains a "perfect" sensor, in the sense that both the presence and the absence of plates, as well as a defect can be safely indicated: Let (x, y) be the values emitted by a combination of a photo electric sensor (x) and an electro-magnetic sensor (y).

Then three situation can arise : $\begin{cases} (\mathsf{false}, \mathsf{false}) & \text{, no plate} \\ (\mathsf{true}, \mathsf{true}) & \text{, plate} \\ (\mathsf{true}, \mathsf{false}) & \text{, at least one sensor defect} \end{cases}$.

Such combined sensors simplify the fault detection, but are not used in practise for cost reasons. Therefore they were not considered in the case study.

REMARK 3. In case of the sensors s2 and s3, there is a kind of combination of two sensors that barely makes sense. Although the two sensors follow each other immediately in the workflow, s3 is more important than s2 in that it also supervises the transfer from the feedbelt to the table. Furthermore, sensor s2 might fail in stopping the feedbelt in case the table is not yet ready for the next plate. However, increasing safety by starting the feedbelt motor only after the table is ready for the next work piece makes sensor s2 superfluous. In order to make the feedbelt safe independent from the table, it would be reasonable to replace s2 by a photo electric cell.

Traffic Lights

According to the task description the traffic lights may not fail. Therefore the following trivial specification may serve as a model:

```
Block  Traffic_Light

  Declarations

    Variables

      Write      light   : {green, red}  Init  red

    Actions

      In         Set_Light     : {green, red}

  Transitions

    ‖  Set_Light(light′)

End.
```

Motors

Two kinds of motors occur in the case study: the motors for the belts that may be running, stopped or defect, and the motors moving components in two directions (e.g. left and right or up and down).

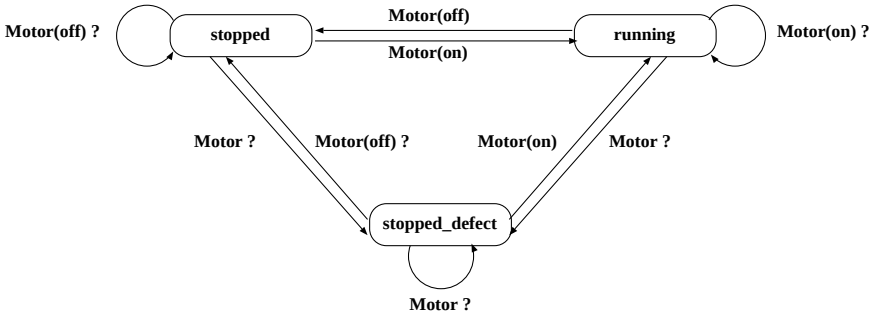


Figure 6.5: BTS corresponding to block Belt_Motor. Motor(off)? abbreviates (Motor(off) $\vee \neg$Motor) which is equivalent to $\neg$Motor(on). Accordingly, Motor? abbreviates (Motor $\vee \neg$Motor) which is equivalent to true.

Block Belt_Motor

Declarations

Variables

Write motor : {stopped, running, stopped_defect} Init stopped

Actions

In Motor : {on, off}

Transitions

$\|$ Motor(on) $\Rightarrow$ motor$'\in$ { running, stopped_defect }

$\|$ Motor(off) $\Rightarrow$ motor$'\in$ { stopped, stopped_defect }

$\|$ true $\longrightarrow$ motor$'$= stopped_defect

$\|$ motor = stopped_defect $\longrightarrow$ motor$'$= stopped

End.

Remark 1. State stopped_defect of the Belt_Motor models both a defect motor and a plate that got stuck.

Similarly, there are motors that may move into two directions:

Block Two_Dirs_Motor

Declarations

Variables

Write motor : {stopped, dir1, dir2, stopped_defect} Init stopped

Actions

In Motor : {dir1, dir2, stop}

Transitions

‖ Motor(dir1) =▷ motor′∈ { dir1, stopped_defect }

‖ Motor(dir2) =▷ motor′∈ { dir2, stopped_defect }

‖ Motor(stop) =▷ motor′∈ { stopped, stopped_defect }

‖ true ⟶ motor′= stopped_defect

‖ motor = stopped_defect ⟶ motor′= stopped

End.

Finally, a specification of the physical unit responsible for movements supervised by a sensor (e.g. the horizontal movement of the table) is given. In this case motor and sensor are modeled together since the "intelligent" sensor is strongly linked to the motor by detecting defects of motor and sensor. Note that parameterization allows one generic block for an arbitrary number of positions (the table has 5 horizontal positions whereas the robot distinguishes 13 positions).

Block Supervised_Motor

Parameters

N : Integer
pos_name : Array(1,N) Of String

Declarations

Variables

Write pos : [1..N]
Write s : String
WriteLocal motor : {dir1, dir2, stop}

Actions

In Motor : {dir1, dir2, stop}

Transitions

‖ Move(motor_hist′)

‖ s ≠ pos_err → Case
‖ motor_hist′= dir1 : pos′= pos ∨ pos > 1 ∧ pos′= pos - 1
‖ motor_hist′= dir2 : pos′= pos ∨ pos < N ∧ pos′= pos + 1
‖ motor_hist′= stop : pos′= pos
End ‖
s′= pos_name(pos′) ∨ s′= pos_err

‖ s = pos_err → s′= pos_name(pos)

Formulas

$\| \bigwedge_{i:[1..N]}$ pos_name(i) $\neq$ pos_err

$\| \bigwedge_{i:[1..N-1]} \Box$ (pos=i $\wedge$ (motor = dir2 Unless pos = i+1) $\wedge \Box\Diamond$(s $\neq$ pos_err)
$\Rightarrow \Diamond$ pos =i+1)

$\| \bigwedge_{i:[2..N]} \Box$ (pos=i $\wedge$ (motor = dir1 Unless pos = i-1) $\wedge \Box\Diamond$(s $\neq$ pos_err)
$\Rightarrow \Diamond$ pos =i-1)

End.

Remark 1. According to the task description, the sensor detects both failures of the motor and failures of itself.

Remark 2. Because the supervised motor has some knowledge about its position, it is possible to state two general liveness properties stating that given it moves continuously from position i towards position $i+1$ ($i-1$) and infinitely often does not fail then eventually it reaches position $i+1$ ($i-1$). These formulas may be refined for concrete instances of the block whenever upper time bounds for the movements are known, as, for example, for the horizontal movement of the table.

The Alarm Signal

The task description of the alarm signal says, that

(1) it should be switched on by the control program(s) if a failure is detected,
(2) it can only be switched off by the user,
(3) it can not fail itself, and
(4) switching off the alarm signal means that all devices are repaired.

Because the alarm signal may not fail, it is sufficient to consider the two states green and red that are visible to the environment. Controllers only report alarms if the alarm light is green and thereby cause a state change to red. Vice Versa, if the alarm light is red, then the user can reset the alarm by means of a Repaired action:

Module Alarm_Signal

Declarations

Variables

Write s21 : {green, red} Init green

Actions

In FB_Alarm, T_Alarm, Robot_Alarm : ()
In Press1_Alarm, Press2_Alarm, DB_Alarm : ()
In Repaired : ()

Abbreviations

Alarm := FB_Alarm $\vee$ T_Alarm $\vee$ Press1_Alarm $\vee$
Press2_Alarm $\vee$ Robot_Alarm $\vee$ DB_Alarm

Transitions

$\|$ { s21 = green } Alarm $\Rightarrow$ s21′= red

$\|$ { s21 = red } Repaired $\Rightarrow$ s21′= green

End.

6.2.2 Modeling the Devices

For brevity, the description of the devices focuses on the feedbelt and the table. This complements the specifications given in [CH95] for the robot and the press of the "original" production cell.

To model the devices, the blocks specifying the sensors and actuators have to be instantiated appropriately, and their behavior has to be described in the context of the device they are part of.

For example, the (physical) feedbelt contains two Boolean sensors, a traffic light and a belt motor. However, these are not arranged by chance, but depend on and influence each other. Expressed otherwise, the set of possible behavior of the subcomponents of the feedbelt is restricted by their special architecture. For example, plates are transported towards sensor s2 at the end and not in the opposite direction. To relate movement and possible sensor values of the feedbelt, a variable fb_state is introduced. It describes whether and where there are plates. The possible behavior of the feedbelt in accordance to the task description (and physical or causality laws) are formulated in a transition section defining the next step relation and a formulas section listing general temporal properties.

The transitions given below imply e.g. that the number of plates is only increased if the traffic light is green. This is assumed for any environment and thus "fixed" in the description of the feedbelt. On the other hand, the maximum time after which a new plate is delivered depends on the concrete environment and is therefore considered as a part of the system assumptions[2].

The first 4 formulas describe that sensors can detect a plate only in case there is one and that as long as they are not defect, they only are in state not_detecting if there is no plate. The last two assumptions exemplify two ways of describing the progress of the feedbelt: Firstly, if a plate is on the belt when a timer is started and the motor at the end keeps running as long as there is no timeout and the plate has not yet arrived at the end, then a plate will eventually arrive at the end. The second liveness property states that if the motor is in state running infinitely often then each plate on the belt will eventually arrive at the end.

The controller for the feedbelt presented in the next section further restricts possible behavior of the feedbelt. For example, it will only allow one plate on the belt at a time (given that initially there is at most one plate). Thus a property to be proven for the composition of the (physical) feedbelt, its controller and the system assumptions will be

□ fb_state≠several_plates .

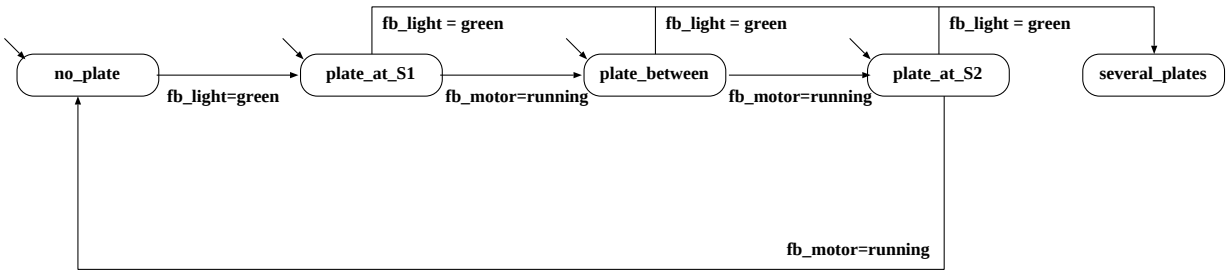


Figure 6.6: Physically possible state changes of the feedbelt.

[2]Without such a time, it would not be possible to distinguish the situation "sensor s1 is defect" from "the environment delivers no more plates"

Module Feedbelt

Include Block Belt_Motor [Motor ← FB_Motor, motor ← fb_motor]

Include Block Sensor [s ← s1, sensor ← sensor1]

Include Block Sensor [s ← s2, sensor ← sensor2]

Include Block Traffic_Light [light ← fb_light, Set_Light ← FB_Set_Light]

Declarations

Variables

Write fb_state : {no_plate, plate_at_S1, plate_between, plate_at_S2, several_plates}

Actions

In StartTimerFB, TimeOutFB : ()

Abbreviations

Plate_Leaves_FB := fb_state = plate_at_S2 ∧ fb_state′= no_plate

Initially

fb_state ≠ several_plates

Transitions

‖ fb_state = no_plate ∧ fb_light = green ⟶ fb_state′= plate_at_S1

‖ fb_state = plate_at_S1 ∧ fb_motor = running ⟶ fb_state′= plate_between

‖ fb_state = plate_between ∧ fb_motor = running ⟶ fb_state′= plate_at_S2

‖ fb_state = plate_at_S2 ∧ fb_motor = running ⟶ fb_state′= no_plate

‖ fb_state ∈ { plate_at_S1, plate_between, plate_at_S2 } ∧ fb_light = green ⟶ fb_state′= several_plates

Formulas

‖ □ (sensor1 = detecting ⇒ fb_state=plate_at_S1)

‖ □ (sensor1 = not_detecting ⇒ fb_state ≠plate_at_S1)

‖ □ (sensor2 = detecting ⇒ fb_state=plate_at_S2)

‖ □ (sensor2 = not_detecting ⇒ fb_state ≠plate_at_S2)

‖ StartTimerFB ∧ (fb_state=plate_at_S1 ∨ fb_state=plate_between) ∧
□ (¬ TimeOutFB ∧ fb_state′≠ plate_at_S2 ⇒ fb_motor = running) ∧
⇒ ◇ fb_state′=plate_at_S2

‖ □◇ fb_motor = running ⇒
(fb_state=plate_at_S1 ∨ fb_state=plate_between ⟼ fb_state=plate_at_S2)

End.

Of course, the modeling could also be done completely in temporal logic. For example, the table might be specified as follows[3]:

Module Table

Include Block Two_Dirs_Motor [motor ← motor_t_v, Motor ← Motor_T_V]

Include Block Supervised_Motor [N ← 5, motor ← motor_t_h, Motor ← Motor_T_H, s ← s6, pos ← pos_t_h, pos_name ← (pos_out, pos_feedbelt, pos_belt2robot, pos_robot, pos_out)]

Include Block Sensor[s ← s3, sensor ← sensor3]

Include Block Sensor[s ← s4, sensor ← sensor4]

Include Block Sensor[s ← s5, sensor ← sensor5]

Declarations

Variables

Write	state	: {no_plate, one_plate, several_plates}
Write	pos_t_v	: {bottom, between, top}

Actions

In	StartTimerTable, TimeOutTable	: ()

Abbreviations

Plate_Arrives_at_Table := state = no_plate ∧ state′= one_plate ∨
state ≠ no_plate ∧ state′= several_plates

Plate_Leaves_Table := state = one_plate ∧ state′= no_plate ∨
state = several_plates ∧ state′≠ several_plates

Assumptions

[A0] pos_t_h ≠ pos_out

[A1] □ pos_t_v′≠ pos_t_v
⇒ (pos_t_v = bottom ∧ motor_t_v = up ∧ pos_t_v′= middle ∨
pos_t_v = middle ∧ motor_t_v = up ∧ pos_t_v′= top ∨
pos_t_v = middle ∧ motor_t_v = down ∧ pos_t_v′= bottom ∨
pos_t_v = top ∧ motor_t_v = down ∧ pos_t_v′= middle)

[A2] □ state′≠ state
⇒ (state = no_plate ∧ state′= one_plate ∨
state = one_plate ∧ state′= several_plates ∨
state = several_plates ∧ state′= one_plate ∨
state = top ∧ state′= no_plate)

[A3] □ ((sensor3 = detecting ⇒ state ≠ no_plate) ∧
(sensor3 = not_detecting ⇒ state = no_plate))

[A4] □ ((sensor4 = detecting ⇒ pos_t_v = bottom) ∧
(sensor4 = not_detecting ⇒ pos_t_v ≠ bottom))

---

[3]Actually, I cheated a bit by just translating a set of transitions to logic (resulting in [A1] and [A2]).

[A5] $\Box$ ((sensor5 = detecting $\Rightarrow$ pos_t_v = top) $\wedge$
(sensor5 = not_detecting $\Rightarrow$ pos_t_v $\neq$ top))

[L1] StartTimerTable $\wedge$ Move_T_V(dir2) $\wedge$ Move_T_H(dir2) $\wedge$
$\Box$ ($\neg$ TimeOutTable $\wedge$ pos_t_v $\neq$ top $\Rightarrow$ motor_t_v = dir2) $\wedge$
$\Box$ ($\neg$ TimeOutTable $\wedge$ s6 $\neq$ pos_robot $\Rightarrow$ motor_t_v = dir2 $\wedge$ s6 $\neq$ pos_err)
$\Rightarrow \Diamond$ (pos_t_v$'$= top $\wedge$ s6$'$= pos_robot)

[L2] StartTimerTable $\wedge$ Move_T_V(dir1) $\wedge$ Move_T_H(dir1) $\wedge$
$\Box$ ($\neg$ TimeOutTable $\wedge$ pos_t_v $\neq$ bottom $\Rightarrow$ motor_t_v = dir1) $\wedge$
$\Box$ ($\neg$ TimeOutTable $\wedge$ s6 $\neq$ pos_feedbelt $\Rightarrow$ motor_t_v = dir1 $\wedge$ s6 $\neq$ pos_err)
$\Rightarrow \Diamond$ (pos_t_v$'$= bottom $\wedge$ s6$'$= pos_feedbelt)

End.

The instance of the supervised motor used in module Table is visualized in Figure 6.7.

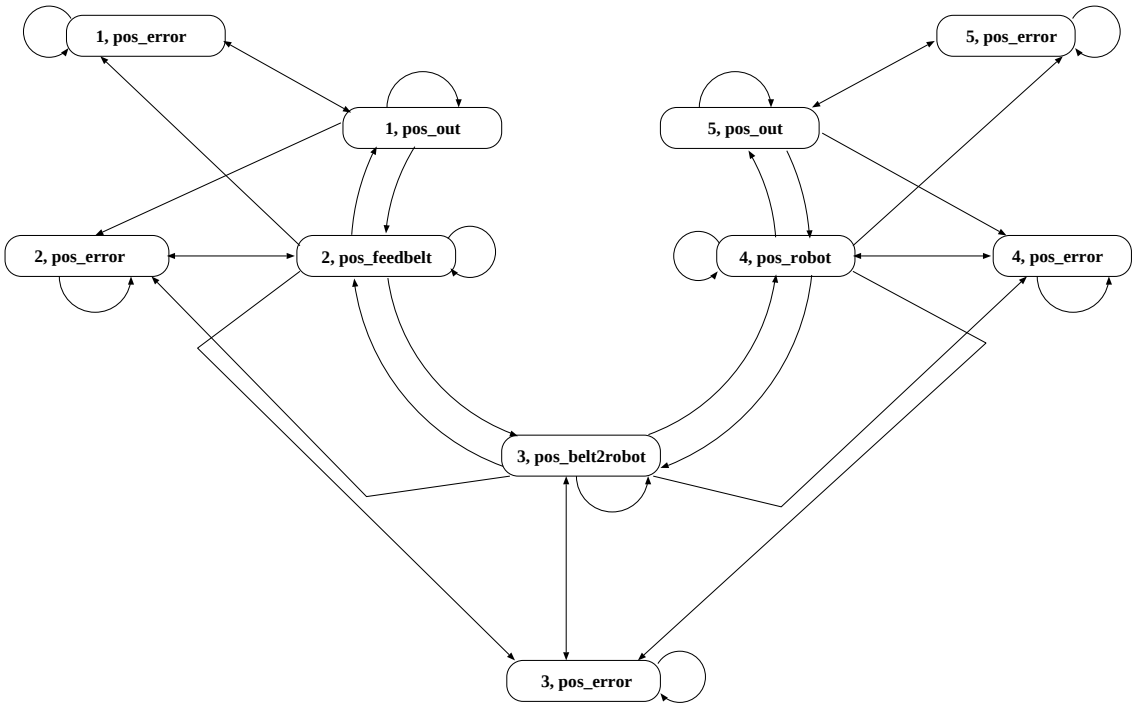


Figure 6.7: All possible state changes of the motor responsible for the horizontal movement of the table (given as pairs (pos_t_h, s6)).

6.2.3 Modeling the Production Cell

To model the complete production cell, the same procedure as for the devices is necessary: Firstly, the modules specifying the devices have to be instantiated appropriately. Secondly, the behavior of the devices has to be related in the context of the production cell. For example, a plate can only "arrive" at the table if a plate "leaves" the feedbelt. The opposite is not true in general, since plates may be lost. However, to prove progress, it is assumed that not all transfers end up with a lost plate.

Besides, the system assumptions have to be specified. Hereby, the developer is required to decide which properties should be encapsulated within the modules and which ones should be part of the system assumptions. The following examples may help in clarifying the distinction between what is part of the specification of physical devices and what belongs to the specification of system assumptions:

Properties concerning physical devices should hold in any environment in order to increase the reuse of the modules. Typically, they are local to the module:

1. A running motor finally reaches a certain position (a liveness property).
2. A device continuously moved by a motor towards position X, finally reaches this position (a liveness property).
3. At most one sensor at a time can signal "press at position X". Although the three sensors indicating bottom, middle and top position are independent, it is physically impossible that a press is in more than one position at a time (an invariant).

Typical system assumptions depend on the environment or the concrete architecture. Often, they are global or temporary:

1. Initially a device is in a certain (controllable) state
   For example, the table's angle initially is somewhere between its position towards the feedbelt and the position towards the robot.
2. Hypotheses concerning the operating system or hardware (considered as part of the environment, that is, not explicitly modeled)
   For example, no transmission errors in some message passing layer (in the case study, this is assumed implicitly simply by not modeling a message passing layer) .
3. If the light at the feedbelt shows green, then within some fixed maximum time a user will deliver a new plate.

Excerpt of the system description:

System Fault_Tolerant_Production_Cell

Layer 2:

⋮

Layer 1:

Include Module Feedbelt

Include Module Table

Include Module $\text{Press}_{[\text{state\_press} \leftarrow \text{state\_press1},\ \text{pos\_press} \leftarrow \text{pos\_press1}]}$

Include Module $\text{Press}_{[\text{state\_press} \leftarrow \text{state\_press2},\ \text{pos\_press} \leftarrow \text{pos\_press2}]}$

Include Module Robot

Include Module Depositbelt

Include Module Alarm_Signal

Systemassumptions

‖ $\Box$ Plate_Arrives_at_T $\Rightarrow$ Plate_Leaves_FB

‖ ($\Box$ $\Diamond$ Plate_Leaves_FB) $\Rightarrow$ ($\Box$ $\Diamond$ Plate_Arrives_at_T)

‖ $\Box$ (Arm1_Picks_Up_Plate_From_T $\Rightarrow$ Plate_Leaves_T)

‖ ($\Box$ $\Diamond$ Plate_Leaves_T) $\Rightarrow$ ($\Box$ $\Diamond$ Arm1_Picks_Up_Plate_From_T)

‖ fb_state=plate_at_S1 Until fb_state=plate_at_S1 $\wedge$ fb_light = red

‖ fb_motor $\neq$ running

⋮

End.

6.3 The Specification of Controllers

6.3.1 The Supervised Transfer Protocol

Any plate that runs through the production cell is transferred 5 times from one device to the next. The following table sums up these transfers:

Source	Target	Active	Supervision	Remarks
environment	feed belt	environment	feed belt (s1)	special case
feed belt	table	feed belt	table (s3)	
table	robot	robot	table (s3)	restricted supervision
robot	press	robot	press (s7 or s11)	
press	robot	robot	press (s7 or s11)	restricted supervision
robot	deposit belt	robot	deposit belt (s18)	
deposit belt	environment	deposit belt (s19)	environment	special case

For all transfers, one of the two devices plays the active role of dropping or picking up the plate, whereas the second device can make use of a sensor to detect whether the transfer was successful. This suggests a simple protocol proceeding in three phases:

1. The passive device signals its readiness to take part in the transfer by sending Ready_For_Transfer.
2. As soon as the active device also is ready, it sends Transfer_Started and begins handing over the plate.
3. On receiving Transfer_Started, the passive device starts a timer. At latest at time-out, it uses its sensor to decide whether the transfer was successful. In case the sensor indicates that the plate was handed over, the controller sends Transfer_Completed(success) immediately. In case of a time-out, both a sensor failure or the loss of the plate (or both) might have occurred. Therefore, the decision whether to send Transfer_Completed(success) or Transfer_Completed(fail) is postponed until error recovery is signaled by the alarm light. At that time, the sensor values are assumed to be trustworthy (see next section) and thus allow a safe decision concerning the outcome of the transfer.

Unfortunately, this strategy has a flaw for the transfers towards the robot, because the sensors at the table (s3) and the press (s7 or s11) might break down during transfer, that is, exactly when they are expected to change their value from true to false. If, for example, s3 breaks down just when the robot tries to grasp the plate, then both the table and the robot leave the transfer point, although it is not guaranteed that the robot has picked up the plate. If the plate can not be delivered to the press (which can be supervised by the press), then either the plate is still on the table and s3 is defect, or the plate was lost by the robot, or the sensor of the press is defect. Due to the first case, the table controller may not signal Ready_For_Transfer to the feedbelt controller before it gets acknowledged whether the plate was successfully transfered to the press or not.

```
Interface  TransferInterface
  Declarations
    Variables
      History    h       : { busy, ready, transfer }  Init  busy
    Actions
      In         Ready_For_Transfer     : ()
      Out        Transfer_Started       : ()
      In         Transfer_Completed     : {success, fail}
```

Transitions

‖ { h = busy } Ready_For_Transfer $\Rightarrow$ h′= ready

‖ { h = ready } Transfer_Started $\Rightarrow$ h′= transfer

‖ { h = transfer } Transfer_Completed $\Rightarrow$ h′= busy

End

This interface is used 10 times in the case study. For example, the instance used by the feedbelt controller is given by:

Include Interface TransferInterface[Ready_For_Transfer ← T_Reached_FB,
Transfer_Started ← Plate_Leaving_FB,
Transfer_Completed ← Transfer_FB2T]

6.3.2 The Error Handling Strategy

The error handling strategy depends heavily on the failure model. The specification of this failure model may be explicit or implicit. For example, in the task description it is said that

"plates do not fall from belts (besides at the end)."

To express this property explicitly, one has to add some state plates_on_floor to the variable fb_state in the specification of the feedbelt. Because no transitions allows to enter this state, one might deduce the invariant

fb_state $\neq$ plates_on_floor $\Rightarrow$ $\Box$ (fb_state $\neq$ plates_on_floor)

Instead of introducing a state that never is entered only to express a property directly, this state was simply omitted. Similarly, the situation "a plate got stuck on the feedbelt" is not modeled explicitly, besides the fact that this failure might happen according to the task description. However, from the controller's view this situation can not be distinguished from the situation "defect belt motor". Thus, to keep the modeling simple, the former situation is rather considered a possible cause on the basis of which the controller might deduce that the motor is broken.

For the design of the controllers, the following two failure assumptions stated in the task description were crucial:

1. initial steps of the controllers are safe in a sense that all sensors are trustworthy, and
2. the same is true for all steps the controllers take when the alarm signal is reset to green after an error occurred.

This may be expressed explicitly in temporal logic. For example,

‖ $\Box$ (fb_phase $\in$ { limbo_init, limbo, limbo_transport }
$\Rightarrow$ sensor1 $\neq$ defect $\wedge$ sensor2 $\neq$ defect)

‖ $\Box$ (fb_error $\wedge$ s21 = green $\Rightarrow$ sensor1 $\neq$ defect $\wedge$ sensor2 $\neq$ defect)

‖ $\Box$ (t_phase = limbo $\Rightarrow$ sensor3 $\neq$ defect $\wedge$ sensor4 $\neq$ defect $\wedge$
sensor5 $\neq$ defect $\wedge$ s6 $\neq$ pos_err)

$$\| \ \Box \ (\ \text{t\_error} \ \wedge \ \text{s21} = \text{green} \ \Rightarrow \text{sensor3} \neq \text{defect} \ \wedge \ \text{sensor4} \neq \text{defect} \ \wedge \ \text{sensor5} \neq \text{defect} \ \wedge \ \text{s6} \neq \text{pos\_err} \ )$$

$$\vdots$$

From this, the following simple error handling strategy is commonly applied for all controllers:

1. Initially, the controllers may take a "safe" step because they may rely on the sensor values due to the assumptions.
2. Then, the controllers run until they detect an error.
3. The controllers stop any movements and change to error mode. In error mode they wait for the alarm light to be set to green.
4. After the alarm light is reset to green, the controllers do a "safe" step again.

Some notes concerning the error handling strategy:

REMARK 1. Errors are dealt with locally whenever possible. If, for example, a motor of the table breaks down, the table controller does not inform the feedbelt that it is broken. Rather the sending of T_Reached_FB is suppressed. This way, the transfer protocol between the two controllers is independent of (temporary) errors of one of the components.

REMARK 2. The values of the alarm light are ignored whenever components have not detected an error previously. Again, this serves local error recovery.

REMARK 3. The controllers typically work in phases, i.e. they cyclicly perform a set of tasks. The interpretation of incoming signals and thus the error handling of the components depends on the actual phase. (for example, when the table is moving and the sensor s3 changes its value from 1 to 0 indicating that a plate has left the table, then it is concluded that the sensor is broken. In phase deliver_plate, this change could also occur due to the fact that the sensor breaks down. The controller however assumes, that it occurred due to the robot picking up the plate.)

Finally, a complete list of situations is given where local errors can be detected (s is an arbitrary sensor, pl a sensor signaling the existence of a plate):

Error	**Situation from controllers view**	**Detection**	**Examples**
sensor	device stopped, s=1 followed by s=0	immediate	feedbelt waiting for table and plate at end (s2)
			table waiting in top position (s5)
sensor	device moving, pl=1 followed by pl=0	immediate	table moving with plate (s3)
sensor	device stopped, pl=1 expected but pl=0 continuously	timeout	feedbelt waiting for plate from environment (s1)
motor	device moving, s=1 continuously	timeout	feedbelt running, plate at end (s2)
			table moving downwards in top position (s5)
motor or sensor	device moving, s=1 expected, but s=0 continuously	timeout	table moving upwards (s2) (vertical motor,s5)
			feedbelt running, plate between (belt motor,s2)

6.3.3 The Feedbelt Controller

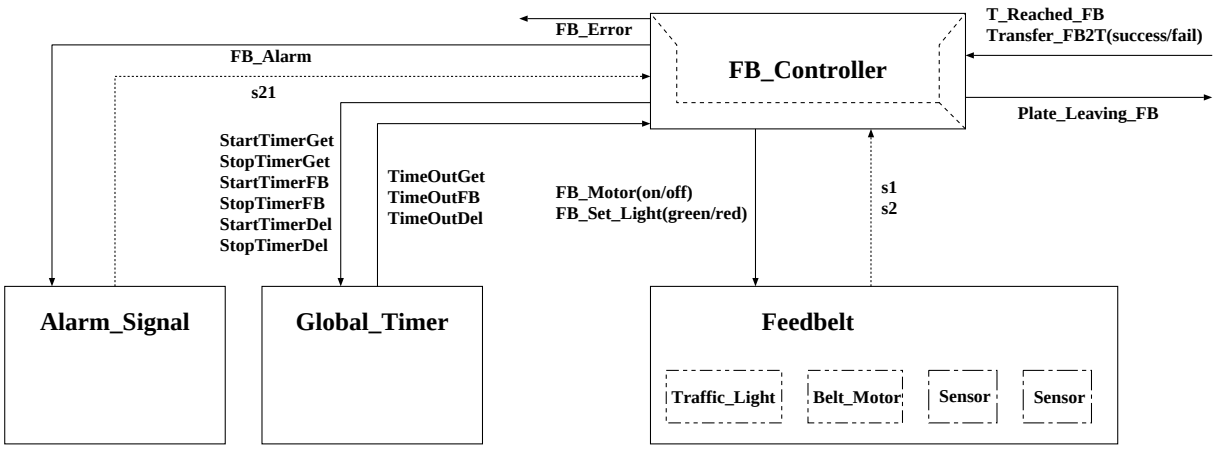


Figure 6.8: Architecture of the Feedbelt

The strategy of the feedbelt controller is best understood looking at the different phases that the controller passes through:

limbo_init: Initially, the controller reads its sensor values which are assumed to be trustworthy. According to these sensor values, all phases besides get_plate are reachable. get_plate is unreachable, because even in case there are no plates at start or at the end, it is still possible that there is a plate between. The controller thus decides to "find out" by changing to state limbo or limbo_transport. Hereby, it is assumed that there is at most one plate on the feed belt initially.

limbo: The controller does not know, whether there is a plate on the belt. Thus, it waits for the table controller signaling that the table is in position pos_feedbelt. Then phase limbo_transport is entered, the belt motor is started and a timer is started

limbo_transport: In this phase, the controller waits for either sensor s2 signaling that a plate has arrived at the end or for a timeout. In case of timeout, the controllers decides that there is no plate and thus changes to get_plate. That is, the controller stops the belt, sets the traffic light to green and starts a timer set to the maximal time the environment needs to deliver the next plate.

get_plate: The controller waits for the environment to deliver a new plate. Now, either sensor s1 indicates that the plate was put on the belt or a time-out signals that sensor s1 must be defect. In the former case, the motor is started together with a timer that gets set to the maximal time a plate needs to reach sensor s2 at the end of the belt. Thereby, phase transport is entered.

transport: The controller waits until either a plate reaches sensor s2 or a timeout signals that either the motor or sensor s2 is defect. If the plate has reached the end without failure, either phase waiting_at_end or phase deliver are entered, depending on the state of the table.

waiting_at_end: Phase waiting_at_end describes the situation where a plate is at the end, but the table is not ready to get it. If sensor s2 breaks during this phase, it is immediately recognized. As soon as the table gets ready, the belt motor is started and phase deliver is entered. In addition, a timer is set that supervises that the motor does not break down while the plate is still at sensor s2.

deliver: This phase is entered with the motor running and the table being ready. If sensor s2 indicates "no plate" then the supervision of the transfer is passed to the table controller by sending Plate_Leaving_FB. Phase deliver is left after the table has send Transfer_FB2T(success). In case of Transfer_FB2T(fail), the controller switches to error mode as is the case for any other error.

The possible changes are visualized in Figure 6.9.

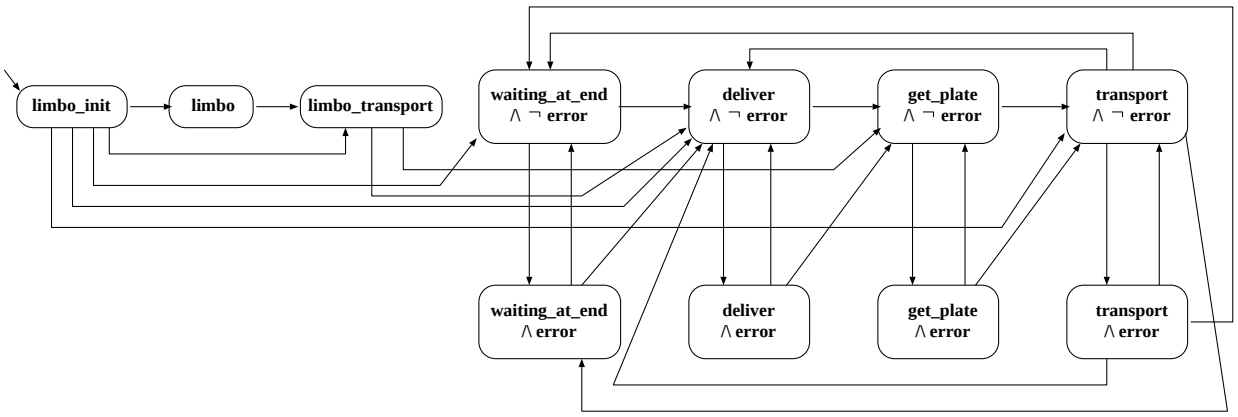


Figure 6.9: The different phases of the feedbelt controller.

Module Feedbelt_Controller

Include Interface TransferInterface[Ready_For_Transfer ← T_Reached_FB,
Transfer_Started ← Plate_Leaving_FB,
Transfer_Completed ← Transfer_FB2T
h ← table]

Declarations

Variables

Read	s1, s2	: Boolean
Read	s21	: {red, green}
Write	fb_error	: Boolean Init false, false
Write	phase	: {limbo_init, limbo, limbo_transport, get_plate, transport, waiting_at_end, deliver_plate} Init limbo_init

Actions

Out	FB_Set_Light	: {red, green}
Out	FB_Motor	: {on, off}
Out	FB_Error	: String
Out	FB_Alarm	: ()
Out	StartTimerFB, StartTimerGet, StartTimerDel	: ()

Out StopTimerFB, StopTimerGet, StopTimerDel : ()
In TimeOutFB, TimeOutGet, TimeOutDel : ()

Transitions

[] phase = limbo_init ∧ s21 = green −→ FB_Set_Light(red) ||
Case
[] s1 : phase′= transport || FB_Motor(on) || StartTimerFB
[] s2 ∧ table = busy : phase′= waiting_at_end || FB_Motor(off)
[] s2 ∧ table = ready : phase′= deliver || FB_Motor(on) || StartTimerDel
[] ¬s1 ∧ ¬s2 ∧ table = busy : phase′= limbo || FB_Motor(off)
[] ¬s1 ∧ ¬s2 ∧ table = ready : phase′= limbo_transport || FB_Motor(on) || StartTimerFB
End

[] phase = limbo ∧ table = ready
−→ phase′= limbo_transport || FB_Motor(on) || StartTimerFB

[] phase = limbo_transport ∧ s2 ∧ table = ready
−→ phase′= deliver || StopTimerFB || StartTimerDel

[] ¬fb_error ∧ phase = get_plate ∧ s1
−→ phase′= transport || FB_Motor(on) || FB_Set_Light(red) || StopTimerGet || StartTimerFB

[] TimeOutGet =▷ If ¬fb_error ∧ phase = get_plate ∧ ¬s1
Then (FB_Set_Light(red) || FB_Error(S1))

[] ¬fb_error ∧ phase = transport −→
Case
[] s2 ∧ table = busy : phase′= waiting_at_end || FB_Motor(off) || StopTimerFB
[] s2 ∧ table = ready : phase′= deliver || StopTimerFB || StartTimerDel
End

[] TimeOutFB =▷
Case
[] phase = limbo_transport ∧ ¬s2 : phase′= get_plate || FB_Motor(off) ||
FB_Set_Light(green) || StartTimerGet
[] ¬fb_error ∧ phase = transport ∧ ¬s2 : FB_Motor(off) || FB_Error(S2_or_motor)
[] ¬fb_error ∧ phase = transport ∧ s1 : FB_Motor(off) || FB_Error(motor)
End

[] ¬fb_error ∧ phase = waiting_at_end
−→ If s2 ∧ table = ready Then (phase′= deliver || FB_Motor(on) || StartTimerDel) ||
If ¬s2 Then (FB_Error(S2))

[] ¬fb_error ∧ phase = deliver −→
Case
[] ¬s2 ∧ table = busy : phase′= get_plate || FB_Motor(off) || StopTimerDel ||
FB_Set_Light(green) || StartTimerGet
[] ¬s2 ∧ table = ready : FB_Motor(off) || StopTimerDel || Plate_Leaving_FB
End

[] TimeOutDel =▷ If ¬fb_error ∧ phase = deliver ∧ s2 Then (FB_Motor(off) || FB_Error(motor))

[] FB_Error ∨ Transfer_FB2T(fail) =▷ fb_error′ || StopTimerGet || StopTimerFB || StopTimerDel

[] fb_error ∧ s21 = green −→ ¬fb_error′ ||
Case
[] phase = get_plate ∧ s1 : phase′= transport || FB_Motor(on) ||
StartTimerFB

```
  ‖ phase = get_plate ∧ ¬s1                  : StartTimerGet || FB_Set_Light(green)
  ‖ phase = transport ∧ ¬s2                  : FB_Motor(on) || StartTimerFB
  ‖ phase = transport ∧ s2 ∧ table = busy    : phase'= waiting_at_end
  ‖ phase = transport ∧ s2 ∧ table = ready   : phase'= deliver || FB_Motor(on) || StartTimerDel
  ‖ phase = waiting_at_end ∧ table = ready   : phase'= deliver || FB_Motor(on) || StartTimerDel
  ‖ phase = deliver ∧ s2                     : FB_Motor(on)
  ‖ phase = deliver ∧ ¬s2 ∧ table = busy     : phase'= get_plate || FB_Set_Light(green) ||
                                               StartTimerGet
End

‖ FB_Error  =▷  If  s21 = green  Then ( FB_Alarm )

End.
```

6.3.4 The Table Controller

The main block of the table controller is organized in phases similar to the feedbelt controller. Two instances of the transfer interface are used to communicate with the feedbelt and the robot controller. The two blocks Feedbelt_History and Robot_History are projections of these instances of the transfer interface that additionally start a timer to supervise the transfer of the plate. Besides, the block Motor_History supervises the movement of the table. Especially, the vertical movement is supervised by starting a timer each time Motor_T_V(dir1) or Motor_T_V(dir2) is emitted.

The phases of the table controller are similar to those of the feedbelt controller. Initially or in case of reset, the controller may rely on the sensor values and can immediately decide which of the other phases to enter. In phase get_plate, the delivering of a plate from the feedbelt is supervised by sensor s3 and timer StartTimerFB2T. After a successful transfer, the table moves towards the robot (phase transport) and, by arriving there, sends T_Reached_R to the robot controller and enters phase deliver_plate. If the sensor s3 signals "no plate" after the robot started to pick up the plate, then the movement back to the feedbelt is started (phase move_back). On reaching the feedbelt, phase wait_for_press is entered. If the robot already has signaled that the plate was successfully passed to one of the presses, then phase get_plate can be entered immediately. Otherwise, the acknowledgment of either success or failure of this transfer to the press is waited for. In the latter case, the controller enters error mode instead of phase get_plate.

```
Module  Table_Controller

  Include  Inverted  Interface  TransferInterface[Ready_For_Transfer ← T_Reached_FB,
                                                 Transfer_Started ← Plate_Leaving_FB,
                                                 Transfer_Completed ← Transfer_FB2T]
  Include  Inverted  Interface  TransferInterface[Ready_For_Transfer ← T_Reached_R,
                                                 Transfer_Started ← Plate_Leaving_T,
                                                 Transfer_Completed ← Transfer_T2R]
  Block  Feedbelt_History
    Declarations
      Variables
```

History h_transferFromFB : {started, finished} Init finished
Local t_transfer_FB2T_start : Real

Transitions

[] { h_transferFromFB = finished } Plate_Leaving_FB
=▷
StartTimerFB2T || h_transferFromFB′= started

[] { h_transferFromFB = started } Transfer_FB2T
=▷
StopTimerFB2T || _transferFromFB′= finished

End

Block Robot_History

Declarations

Variables

History h_transferToR : {started, finished} Init finished
Local t_transfer_T2R_start : Real

Transitions

[] { h_transferToR = finished } Plate_Leaving_T
=▷
StartTimerT2R || h_transferToR′= started

[] { h_transferToR = started } Transfer_T2R
=▷
StopTimerT2R || h_transferToR′= finished

End

Block Motor_History

Declarations

Variables

Local moving_v, moving_h : Boolean Init false, false

Actions

Out Motor_T_H : {dir1, dir2, stop}
Out Motor_T_V : {dir2, dir1, stop}

Transitions

[] Motor_T_V(stop) =▷ ¬moving_v′ || StopTimerTable

[] Motor_T_V(dir1) ∨ Motor_T_V(dir2) =▷ moving_v′ || StartTimerTable

[] Motor_T_H(stop) =▷ ¬moving_h′

[] Motor_T_H(dir1) ∨ Motor_T_H(dir2) =▷ moving_h′

End

Declarations

Variables

Read	s3, s4, s5	: Boolean
Read	s21	: {red, green}
Read	s6	: String
Read	t	: Real
Write	t_error	: Boolean Init false
Write	phase	: {limbo, get_plate, transport, deliver_plate, move_back, wait_for_press } Init limbo
Local	plate_reached_press	: Boolean

Actions

Out	T_Alarm	: ()
Out	T_Error	: String
In	Transfer2Press	: { success, fail }
Out	StartTimerFB2T, StartTimerT2R, StartTimerTable	: ()
Out	StopTimerFB2T, StopTimerT2R, StopTimerTable	: ()
In	TimeOutFB2T, TimeOutT2R, TimeOutTable	: ()

Transitions

$[\!]$ phase = limbo $\vee$ t_error $\wedge$ s21 = green
$\longrightarrow$
¬t_error′ ||
Case
- $[\!]$ ¬s3 $\wedge$ s4 $\wedge$ s6 = pos_feedbelt : If t_error Then(phase′= wait_for_press) Else(phase′= get_plate || T_Reached_FB) || Motor_T_H(stop) || Motor_T_V(stop)
- $[\!]$ ¬s3 $\wedge$ s4 $\wedge$ s6 ≠ pos_feedbelt : phase′= move_back || Motor_T_H(dir1) || Motor_T_V(stop)
- $[\!]$ ¬s3 $\wedge$ ¬s4 $\wedge$ s6 = pos_feedbelt : phase′= move_back || Motor_T_H(stop) || Motor_T_V(dir1)
- $[\!]$ ¬s3 $\wedge$ ¬s4 $\wedge$ s6 ≠ pos_feedbelt : phase′= move_back || Motor_T_H(dir1) || Motor_T_V(dir1)
- $[\!]$ s3 $\wedge$ s5 $\wedge$ s6 = pos_robot : phase′= deliver_plate || T_Reached_R || Motor_T_H(stop) || Motor_T_V(stop)
- $[\!]$ s3 $\wedge$ s5 $\wedge$ s6 ≠ pos_robot : phase′= transport || Motor_T_H(dir2) || Motor_T_V(stop)
- $[\!]$ s3 $\wedge$ ¬s5 $\wedge$ s6 = pos_robot : phase′= transport || Motor_T_H(stop) || Motor_T_V(dir2)
- $[\!]$ s3 $\wedge$ ¬s5 $\wedge$ s6 ≠ pos_robot : phase′= transport || Motor_T_H(dir2) || Motor_T_V(dir2)

End

$[\!]$ ¬t_error $\wedge$ phase = get_plate $\wedge$ s3
$\longrightarrow$ phase′= transport || Motor_T_H(dir2) || Motor_T_V(dir2) || Transfer_FB2T(success)

$[\!]$ TimeOutFB2T $=\triangleright$ If ¬t_error $\wedge$ phase = get_plate $\wedge$ ¬s3 $\wedge$ h_transferFromFB = started Then (T_Error(S2_and_FB_motor_or_loss_or_S3) || Transfer_FB2T(fail))

$[\!]$ ¬t_error $\wedge$ phase = transport $\wedge$ s3 $\wedge$ s6 ≠ pos_err

```
   ⟶
   Case
     ▯ s5 ∧ moving_v                          : Motor_T_V(stop)
     ▯ s6 = pos_robot ∧ moving_h              : Motor_T_H(stop)
   End ||
   If      s5 ∧ s6 = pos_robot   Then( phase′= deliver_plate || T_Reached_R )
▯  ¬t_error ∧ phase = transport ∧ ¬ s3 ∧ s6 ≠ pos_err  ⟶  T_Error(S3)
▯  ¬t_error ∧ phase = transport ∧ ¬ s3 ∧ s6 = pos_err  ⟶  T_Error(S3_and_MS6)
▯  ¬t_error ∧ phase = transport ∧ s3 ∧ s6 = pos_err  ⟶  T_Error(MS6)
▯  TimeOutTable  =▷
   Case
     ▯ ¬t_error ∧ phase = transport ∧ s3 ∧ s6 ≠ pos_err : T_Error(S5_or_V_Motor)
     ▯ ¬t_error ∧ phase = transport ∧ s3 ∧ s6 = pos_err : T_Error(S5_or_V_Motor_AND_MS6)
     ▯ ¬t_error ∧ phase = transport ∧¬s3 ∧ s6 ≠ pos_err : T_Error(S3_AND_S5_or_V_Motor)
     ▯ ¬t_error ∧ phase = transport ∧¬s3 ∧ s6 = pos_err : T_Error(S3_AND_S5_or_V_Motor_AND_MS6)
   End

▯  ¬t_error ∧ phase = deliver_plate ∧ h_transferToR = started ∧ ¬s3
   ⟶ phase′= move_back || Motor_T_H(dir1) || Motor_T_V(dir1) || Transfer_T2R(success)
▯  TimeOutT2R  =▷  If ¬t_error ∧ phase = deliver_plate ∧ h_transferToR = started ∧ s3
                   Then ( T_Error(Plate_not_picked_up) || Transfer_T2R(fail) )
▯  ¬t_error ∧ phase = move_back ∧ s6 ≠ pos_err
   ⟶
   Case
     ▯ s4 ∧ moving_v                          : Motor_T_V(stop)
     ▯ s6 = pos_feedbelt ∧ moving_h           : Motor_T_H(stop)
   End ||
   If      s4 ∧ s6 = pos_feedbelt ∧   Then( phase′= wait_for_press )
▯  Transfer2Press(success)  =▷  plate_reached_press′
▯  ¬t_error ∧ phase = wait_for_press ∧ plate_reached_press
   ⟶ ¬plate_reached_press′ || phase′= get_plate || T_Reached_FB
▯  ¬t_error ∧ phase = move_back ∧ s6 = pos_err  ⟶  T_Error(MS6)
▯  TimeOutTable  =▷
   Case
     ▯ ¬t_error ∧ phase = move_back ∧ s6 ≠ pos_err  : T_Error(S4_or_V_Motor)
     ▯ ¬t_error ∧ phase = move_back ∧ s6 = pos_err  : T_Error(S4_or_V_Motor_AND_MS6)
   End

▯  T_Error ∨ Transfer2Press(fail)  =▷ t_error′ || If s21 = green Then ( T_Alarm ) ||
                                       If moving_v Then ( Motor_T_V(stop) ) ||
                                       If moving_h Then ( Motor_T_H(stop) )

End.
```

6.3.5 The Complete System

The complete system is obtained by instantiating the modules according to the concrete layout of the case study:

System Fault_Tolerant_Production_Cell

Layer 2:

Include Module Feedbelt_Controller[phase ← fb_phase]

Include Module Table_Controller[phase ← t_phase]

Include Module Press_Controller[phase ← p1_phase, P_Motor ← P1_Motor
s_top ← s10, s_middle ← s9, s_bottom ← s8, s_plate ← s7]

Include Module Press_Controller[phase ← p2_phase, P_Motor ← P2_Motor
s_top ← s14, s_middle ← s13, s_bottom ← s12, s_plate ← s11]

Include Module Robot_Controller[phase ← r_phase]

Include Module Depositbelt_Controller[phase ← r_phase]

Layer 1:

Include Module Feedbelt

Include Module Table

Include Module Press[state_press ← state_press1, pos_press ← pos_press1, P_Motor ← P1_Motor
s_top ← s10, s_middle ← s9, s_bottom ← s8, s_plate ← s7]

Include Module Press[state_press ← state_press2, pos_press ← pos_press2, P_Motor ← P2_Motor
s_top ← s14, s_middle ← s13, s_bottom ← s12, s_plate ← s11]

Include Module Robot

Include Module Depositbelt

Include Module Alarm_Signal

Systemassumptions

‖ □ Plate_Arrives_at_T ⇒ Plate_Leaves_FB

‖ (□ ◇ Plate_Leaves_FB) ⇒ (□ ◇ Plate_Arrives_at_T)

‖ □ (Arm1_Picks_Up_Plate_From_T ⇒ Plate_Leaves_T)

‖ (□ ◇ Plate_Leaves_T) ⇒ (□ ◇ Arm1_Picks_Up_Plate_From_T)

‖ □ (fb_state=plate_at_S1 ∧ fb_light = green ⇒ fb_state′= plate_at_S1)

‖ fb_motor ≠ running

‖ □ (fb_phase ∈ { limbo_init, limbo, limbo_transport } ∨ fb_error ∧ s21 = green
⇒ sensor1≠defect ∧ sensor2≠defect ∧ fb_motor≠defect)

‖ □ (t_phase = limbo ∨ t_error ∧ s21 = green
⇒ sensor3 ≠ defect ∧ sensor4 ≠ defect ∧ sensor5 ≠ defect ∧ s6 ≠ pos_err)

End.

Only one of the system assumptions needs further explanation: In the model of the feedbelt, the state several_plates can be entered only if the traffic light at the beginning of the belt shows green. However, this is not sufficient to disallow several plates on the belt because the environment could place several plates onto the belt within one green phase. To avoid this, the system assumption

$$\Box \ (\ \text{fb\_state}=\text{plate\_at\_S1} \wedge \text{fb\_light} = \text{green} \quad \Rightarrow \quad \text{fb\_state}'= \text{plate\_at\_S1}\ )$$

is added. For the states plate_between and plate_at_S1, the feedbelt controller takes care that there is no green light.

6.4 Compilation

In [CHB96b], a tool is described that allows to compile TLT specifications to the language C and the supporting library MPI (for *Message Passing Interface*). MPI is a public-domain library for message passing in a distributed environment [GLS94]. In order to implement the specifications using this tool, the specifications for the controllers as presented above have to be changed to some extend:

1. MPI does not support enumeration types. Thus all enumeration types got coded as integers.
2. For the same reason, variables must be coded as actions.
3. One additional TLT module is needed to gather the inputs and outputs to the simulation of the production cell.
4. Since the transmission times (over the LAN) far exceed cpu times necessary to compute the control values, all transmissions of the controllers to this extra module need to be packed together wherever possible.

6.5 Verification

In this section, some samples of verification tasks are presented. The proofs for these samples make use of the intuitiveness of (abstract) Boolean transition systems wherever possible.

As an example of a more demanding invariant, it will be proven that

$$\Box\ \text{fb\_state} \neq \text{several\_plates}$$

holds for the production cell. That is, strictly speaking, it is shown that

$$\Phi(\text{Fault\_Tolerant\_Production\_Cell}) \ \Rightarrow \ \Box\ \text{fb\_state} \neq \text{several\_plates}$$

but for simplicity, the premise Φ(Fault_Tolerant_Production_Cell) is omitted in all formulas of this section.

Intuitively, the property holds because it is assumed to hold initially, and afterwards the feedbelt controller takes care not to allow a second plate on the belt.

That the property holds initially follows immediately from the feedbelt as modeled in Section 6.2.2. Now assume there is a transition that violates the property, that is,

$$\Diamond\ (\ \text{fb\_state} \neq \text{several\_plates} \wedge \text{fb\_state}' = \text{several\_plates}\ )$$

From the programming text of the feedbelt model or from the BTS representation in Figure 6.6, it is also obvious that the state fb_state = several_plates is entered only in case the traffic light at the beginning of the feedbelt shows a green light. That is,

(1) □ (fb_state ≠several_plates ∧ fb_state′= several_plates ⇒ fb_light = green)

Figure 6.10: On top, the BTS for the traffic light is shown. In the middle, there is an abstract BTS for the feedbelt controller. In the product of the upper two BTS, the state $\mathsf{fb\_light} = \mathsf{green} \wedge \mathsf{fb\_phase} \neq \mathsf{get\_plate}$ is not reachable.

Taking a look at the feedbelt controller, the traffic traffic light is set to green only if phase get_plate is entered. The abstract BTS in the middle of Figure 6.10 captures this situation. The behavior of the traffic light is given at the top of that figure. In the product of the two BTS as shown at the bottom, only three states are reachable, and indeed

(2) □ (fb_light = green ⇒ fb_phase = get_plate)

That is, together with (1),

(3) □ (fb_state ≠several_plates ∧ fb_state′= several_plates ⇒ fb_phase = get_plate)

Next, the following invariant may be deduced from Figure 6.11,

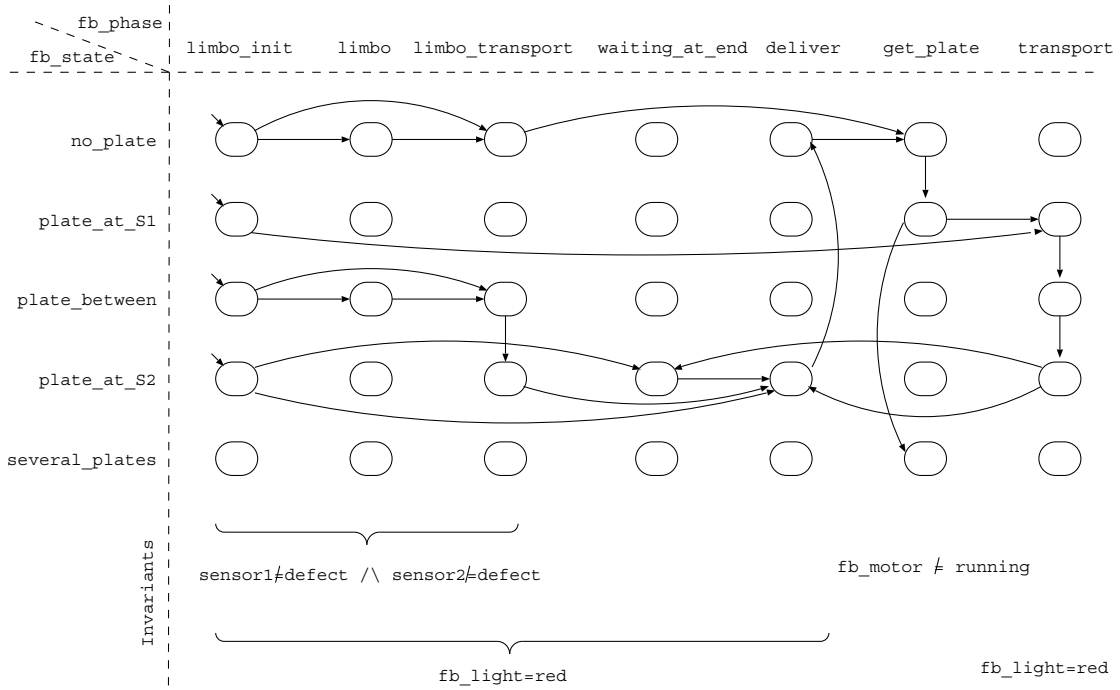


Figure 6.11: The product of the feedbelt and the feedbelt controller with respect to the variables fb_state and fb_phase. Only transitions leaving reachable states are shown. On the bottom, some invariants used to reduce the number of transitions are given. Of these invariants, $\Box(\mathsf{fb\_phase} \in \{\mathsf{limbo\_init}, \mathsf{limbo}, \mathsf{limbo\_transport}\} \Rightarrow \mathsf{sensor1} \neq \mathsf{defect} \wedge \mathsf{sensor2} \neq \mathsf{defect})$ is a system invariant, whereas the other two ones follow from Figure 6.10 and Figure 6.13 respectively.

(4) $\Box$ (fb_phase = get_plate $\Rightarrow$ fb_state $\neq$ plate_between $\wedge$ fb_state $\neq$ plate_at_S2)

That is, together with (3) and the model of the feedbelt,

(5) $\Box$ (fb_state $\neq$ several_plates $\wedge$ fb_state$'$= several_plates
$\Rightarrow$ fb_state = plate_at_S1)

Together with (1),

(5) $\Box$ (fb_state $\neq$several_plates $\wedge$ fb_state$'$= several_plates
$\Rightarrow$ fb_light = green $\wedge$ fb_state = plate_at_S1)

But this contradicts the system assumption

fb_state=plate_at_S1 Until fb_state=plate_at_S1 $\wedge$ fb_light = red

■

Mostly, invariants are easier to prove. For example, the invariant

$\Box$ (Transfer_FB2T $\Rightarrow$ fb_state = deliver)

can be proven directly by use of one suitable abstract BTS as shown in Figure 6.12.

As a last example, the invariant

$\Box$ (fb_error $\Rightarrow$ fb_motor $\neq$ running)

is easily deduced from the invariant

□ (fb_motor = running ⇒ fb_phase ∈ { limbo_transport, deliver, transport })

that restricts the phases in which the motor might be running (this invariant is validated by Figure 6.13): Initially, ¬fb_error and thus the invariant holds. But immediately from the code of the feedbelt controller, one can verify that it is not possible to enter an error state in one of the phases in which the motor might be running without stopping the motor:

□ (¬ fb_error ∧ fb_error′ ∧ fb_phase′ ∈ { limbo_transport, deliver, transport }
⇒ FB_Motor(off))

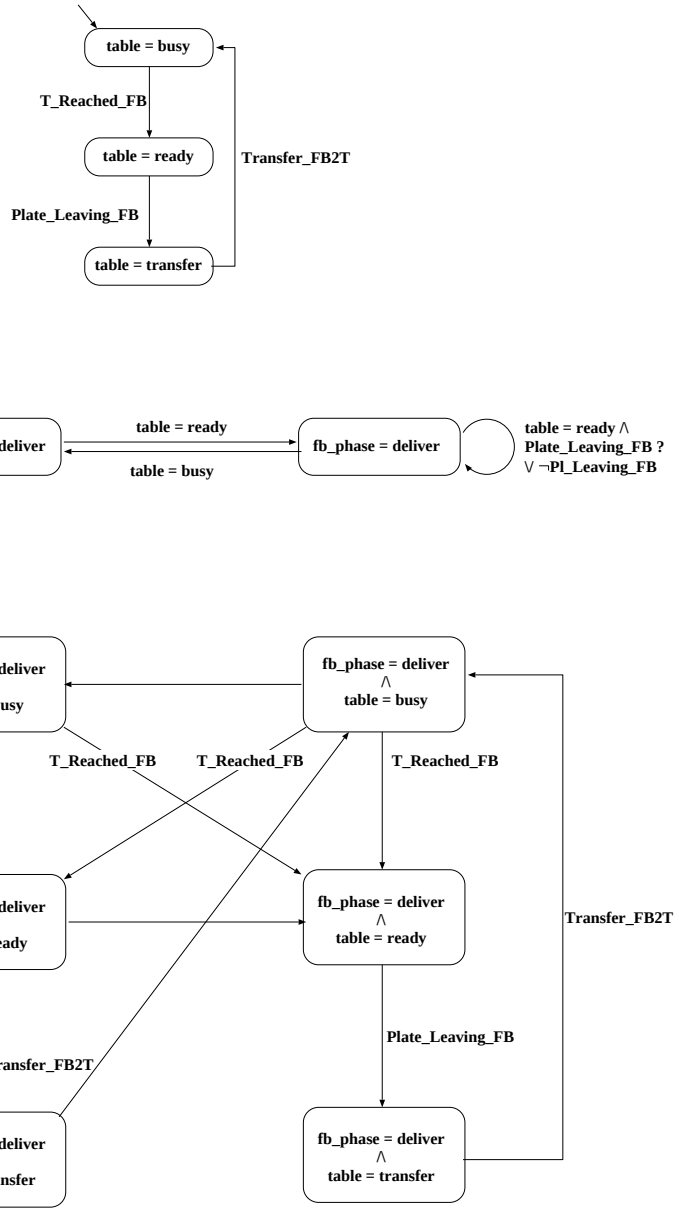


Figure 6.12: At the bottom, one possible abstract BTS to validate the invariant □(Transfer_FB2T ⇒ fb_state = deliver) is given. It results from taking the product of the transfer interface (on top) with a suitable abstraction of the feedbelt controller (middle).

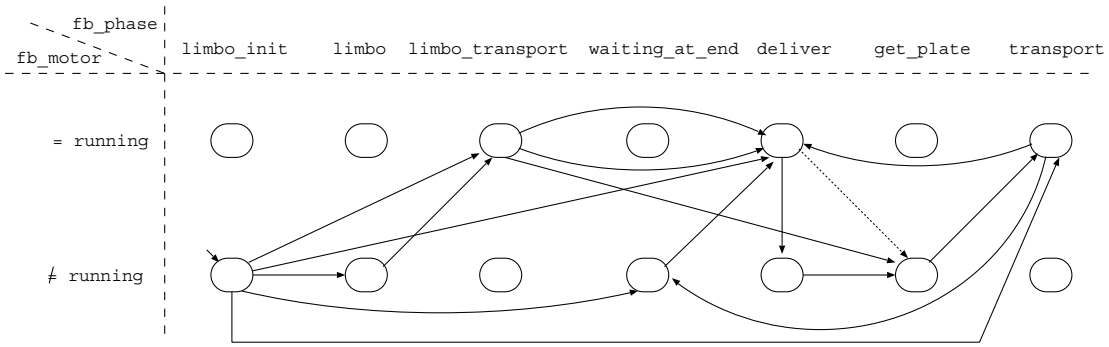


Figure 6.13: The product of the feedbelt motor and the feedbelt controller with respect to the variables fb_motor and fb_phase. Only transitions leaving reachable states are shown. From the product, it follows especially that $\Box(\mathsf{fb\_phase} = \mathsf{get\_plate} \Rightarrow \mathsf{fb\_motor} \neq \mathsf{running})$ holds for the production cell.

Chapter 7

Conclusion and Further Work

The temporal language of transitions is a modular specification language that combines the expressiveness of synchronous languages (like Signal or Esterel) with the expressiveness of state based languages (like UNITY). The programming view allows to describe the behavior of modules in an operational style using two kinds of transitions. The interaction of the modules can be formulated with assumption-commitment protocols in interfaces. The programming view is linked to a linear temporal logic and to an automata format. To achieve a direct correspondence of the main language constructs, both logic and automata are extended to deal with actions and transitions. The logic also gets used for the consistency conditions and properties of the modules. Finite automata can be extracted directly from the programming notation. Parameterization eases to deal with repeating structures.

However, the framework presented in this thesis is too general to be used in nowadays highly specialized software business. The two main obstacles hereby are that

1. some concepts like the notion of fairness are too demanding and too "new" for many users in a special application domain, and conversely

2. domain specific concepts the users are familiar with are missing.

For short, TLT needs to be adjusted to specific domains. The remainder of this thesis gives some hints of how this could be achieved in the domain of control systems.

7.1 Relationship to IEC 1131/IEC 1499

Until recently, control systems were programmed almost exclusively using either (relay) ladder diagrams or instruction lists (an assembly language). Even with the emergence of the programming standard IEC 1131 in 1992, the programming of control systems was restricted to a paradigm that only knew variables as a communication means.

A typical control application in the context of IEC 1131 is captured by the following continuous function chart:

The scheduling of such an application can be described as a TLT specification as follows:

$[\!]$ Schedule_App1 $=\triangleright$ v'=FB1(x) $\wedge$ w'=FB2(y,v') $\wedge$ z'=FB3(v',w')

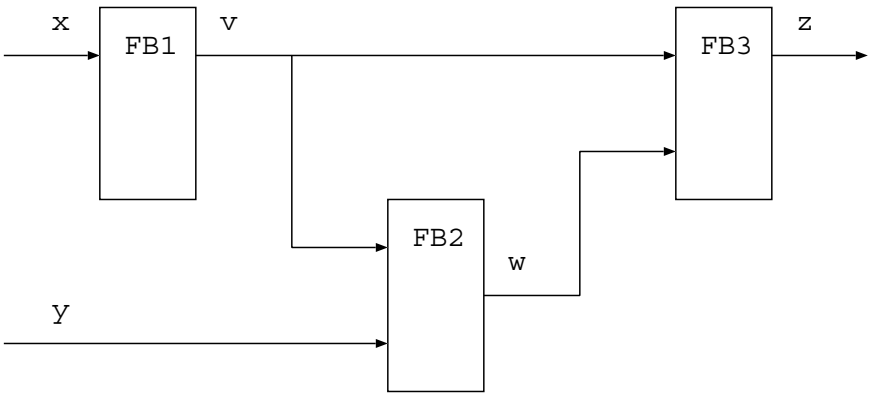


Figure 7.1: Contionuous function charts timers or adders.

where Schedule_App1 is caused by the (real time) operating system of a PLC (typically, some programs are scheduled cyclically whereas others get scheduled interrupt driven or, e.g. at starting time).

Actually, this representation is already an abstraction. In reality, the execution of a function block is not considered to be atomic (although function blocks mostly are guaranteed to terminate within a given number of processor cycles). Even worse, within function blocks, global variables might be changed (IEC 1131 function blocks are by no means fully abstract). Thus the execution order may be crucial and has to be defined explicitly. If the execution sequence in the above would differ from the one suggested by the data-flow and be, say FB2, FB1, FB3, then the scheduling of the program might be specified as

$$[\![\ \text{Schedule\_App1} \ \Rightarrow\!\triangleright\ \ v'=\text{FB1}(x) \wedge w'=\text{FB2}(y,v) \wedge z'=\text{FB3}(v',w') \quad .$$

That is, function block FB2 relies on the old value of the variable v.

The main disadvantage of the IEC 1131 is, however, that in order to realize communication between different programmable logic controllers extra communication function blocks have to be used to simulate events. The use of these communication function blocks is somewhat artificial and makes the IEC 1131 languages not well suited for distributed control systems. This drawback made necessary the definition of a second norm, called IEC 1499. Since July 1997, a first draft of this norm is available. It facilitates communication by allowing the use of signals (in the norm called "events"). The overall model of a function block according to IEC 1499 is shown in Figure 7.2.

IEC 1499 function blocks are divided into an execution control chart and a set of algorithms (typically these are "ordinary" IEC 1131 function blocks). On each incoming signal, a corresponding Boolean valued history variable is set to true. Together with each signal, a set of variables gets sampled, again by updating corresponding history variables. With each state of the control chart, a set of algorithms is associated that get executed if the execution control chart enters the state. On termination of these algorithms, output signals may be sent. Afterwards, the execution control chart changes its state depending on its actual state, the values of the history variables and the values of internal variables of the algorithms.

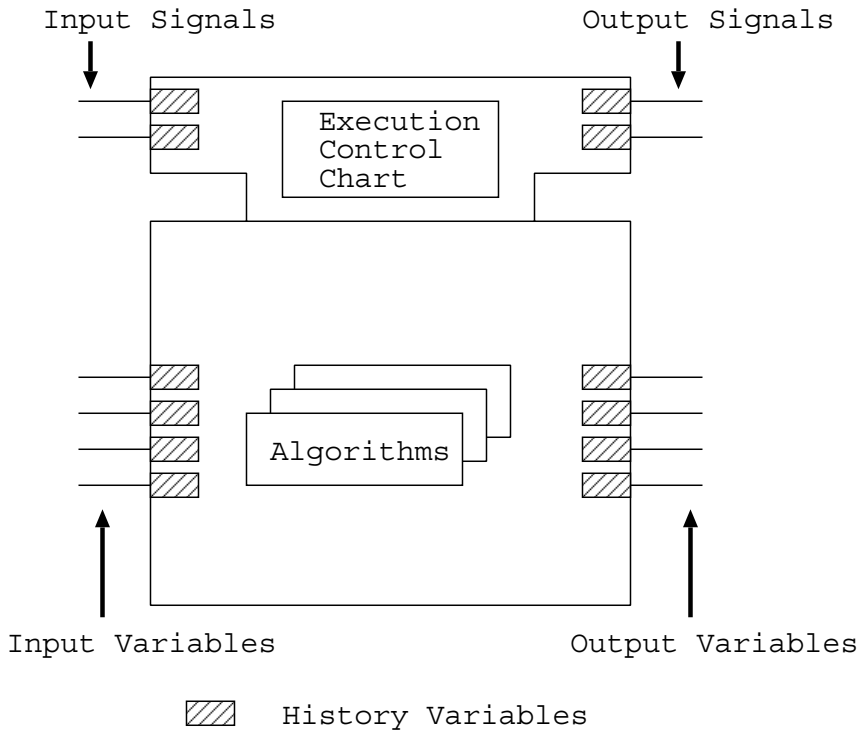


Figure 7.2: An IEC 1499 function block.

The execution control charts used in the norm IEC 1499 can be described easily as TLT specifications. This way, the graphical representation introduced in IEC 1499 is given some formal semantics that may be used to formally reason about the distribution aspects of control applications described that way. Notwithstanding, the local algorithms could still be described as function blocks using the established languages of IEC 1131.

Appendix A

Technical Details

This appendix summarizes some purely technical proofs left out in the main text.

Lemma 5.7.6(Stutter Invariance)
Let ξ *be an arbitrary valuation of* Vars, *and* ξ' *a valuation of* Vars' *such that* $\xi'(x') = \xi(x)$ *for all* $x \in \mathsf{Vars}$.

Then $[\![\delta(\mathsf{block})]\!]^{(\xi,\tau,\xi')}$ *or, equivalently,* $[\![(\bigwedge_{a \in \mathsf{Vars} \cup \mathsf{Acts}} \mathsf{Stutter}(a)) \Rightarrow \delta(\mathsf{block})]\!]$.

Proof:

The lemma is checked for the two conjuncts of $\delta(\mathsf{block})$ separately:

Firstly, suppose the claim does not hold for some i in the first conjunct δ_{env}, i.e.
$[\![\exists_{s_i} (ev_i \wedge \neg assume_i \wedge \neg delay_i \wedge \neg \mathrm{Trl}(cmd_i))]\!]^{(\xi,\tau,\xi')}$.
Then, especially $[\![\exists_{s_i} ev_i]\!]^{(\xi,\tau,\xi')}$. But that contradicts lemma 5.6.2.

Secondly, $[\![v' = v]\!]^{(\xi,\tau,\xi')}$ for all $v \in \mathrm{Ctr}_V(\mathsf{block})$ and $[\![\neg A]\!]^{(\xi,\tau,\xi')}$ for all $A \in \mathrm{Ctr}_{Act}(\mathsf{block})$.
Therefore $[\![\bigwedge_{a \in \mathrm{Ctr}(\mathsf{block})} \mathsf{Stutter}(a)]\!]^{(\xi,\tau,\xi')}$ and thus also $[\![\delta_{ctr}(\mathsf{block})]\!]^{(\xi,\tau,\xi')}$.

■

Lemma 5.7.7(Input Enabledness)
$[\![(\bigwedge_{1 \le i \le |tt|} \forall_{s_i} (ev_i \Rightarrow g_i)) \Rightarrow \exists_{\mathrm{Ctr}'(\mathsf{block})} \delta(\mathsf{block})]\!]$

Proof:

Let (ξ, α, ξ') be an arbitrary valuation. Then two cases are distinguished:

Firstly, assume that $[\![(\bigwedge_{1 \le i \le |tt|} \forall_{s_i} \neg ev_i)]\!]^{(\xi,\alpha,\xi')}$.

Then, define

$$\hat{\alpha}(A) = \begin{cases} \bot & , A \in \mathrm{Ctr}(\mathsf{block}) \\ \alpha(A) & , A \notin \mathrm{Ctr}(\mathsf{block}) \end{cases} \quad \text{and} \quad \hat{\xi}'(x) = \begin{cases} \xi(x) & , x \in Ctr(\mathsf{block}) \\ \xi'(x) & , x \notin Ctr(\mathsf{block}) \end{cases}.$$

Because all events are either In(for which $\hat{\alpha}$ agrees with τ) or Out-events (for which $\hat{\alpha}$ agrees with α), $[\![(\bigwedge_{1\leq i\leq |tt|} \forall_{s_i} \neg ev_i)]\!]^{(\xi,\hat{\alpha},\hat{\xi'})}$ continues to hold. Thus $[\![\delta_{env}(\mathsf{block})]\!]^{(\xi,\hat{\alpha},\hat{\xi'})}$.

Because all controlled variables and actions stutter, $[\![\delta_{ctr}(\mathsf{block})]\!]^{(\xi,\hat{\alpha},\hat{\xi'})}$ holds too. Together, $[\![\delta(\mathsf{block})]\!]^{(\xi,\hat{\alpha},\hat{\xi'})}$ and thus $[\![\exists_{\mathrm{Ctr}'(\mathsf{block})} \delta(\mathsf{block})]\!]^{(\xi,\alpha,\xi')}$.

Now assume $[\![(\bigvee_{1\leq i\leq |tt|} \exists_{s_i} ev_i) \wedge (\bigwedge_{1\leq i\leq |tt|} \forall_{s_i} (ev_i \Rightarrow g_i))]\!]^{(\xi,\alpha,\xi')}$.

Then $[\![\exists_{s_j} (ev_j \wedge g_j) \wedge (\bigwedge_{1\leq i\leq |tt|} \forall_{s_i} (ev_i \Rightarrow g_i))]\!]^{(\xi,\alpha,\xi')}$ for some $1 \leq j \leq |tt|$.

With [TT_Consistency], it follows that $[\![\exists_{s_j} (ev_j \wedge g_j) \wedge (\bigwedge_{\substack{1\leq i\leq |tt| \\ i\neq j}} \neg \exists_{s_i} ev_i)]\!]^{(\xi,\alpha,\xi')}$.

From [TT], it follows that there is a valuation $(\xi, \tilde{\alpha}, \tilde{\xi'})$ with $\tilde{\alpha}(A) = \alpha(A)$ for all $A \notin \mathrm{Ctr}(cmd_j)$, $\tilde{\xi'}(x) = \xi'(x)$ for all $x \notin \mathrm{Ctr}(cmd_j)$ and such that $[\![\exists_{s_j} (ev_j \wedge g_j \wedge \mathrm{Trl}(cmd_j))]\!]^{(\xi,\tilde{\alpha},\tilde{\xi'})}$.

Due to [Ev], also $[\![\bigwedge_{\substack{1\leq i\leq |tt| \\ i\neq j}} \neg \exists_{s_i} (ev_i)]\!]^{(\xi,\tilde{\alpha},\tilde{\xi'})}$

and therefore $[\![(\exists_{s_j} ev_j \wedge g_j \wedge \mathrm{Trl}(cmd_j)) \wedge (\bigwedge_{\substack{1\leq i\leq |tt| \\ i\neq j}} \neg \exists_{s_i} (ev_i))]\!]^{(\xi,\tilde{\alpha},\tilde{\xi'})}$.

Defining $\hat{\alpha}(A) = \begin{cases} \tilde{\alpha}(A) & , A \in \mathrm{Ctr}(cmd_j) \\ \bot & , A \in \mathrm{Ctr}(\mathsf{block}) - \mathrm{Ctr}(cmd_j) \\ \alpha(A) & , \text{else} \end{cases}$

and $\hat{\xi'}(x) = \begin{cases} \tilde{\xi}(x) & , x \in \mathrm{Ctr}(cmd_j) \\ \xi(x) & , x \in Ctr(\mathsf{block}) - \mathrm{Ctr}(cmd_j) \\ \xi'(x) & , \text{else} \end{cases}$,

it holds that $[\![\delta(\mathsf{block})]\!]^{(\xi,\hat{\alpha},\hat{\xi'})}$ and thus $[\![\exists_{\mathrm{Ctr}'(\mathsf{block})} \delta(\mathsf{block})]\!]^{(\xi,\alpha,\xi')}$.

■

Lemma 5.7.10
Local progress may also be expressed explicitly as the fairness conditions

$$< \mathsf{LP}, \mathsf{WF}, \bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|} \mathsf{guard_i}, \bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|} \mathsf{guard_i} \wedge \mathrm{Trl}(\mathsf{cmd_i}) >.$$

Proof:

$$< \mathsf{LP}, \mathsf{WF}, \bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|} \mathsf{guard_i}, \bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|} \mathsf{guard_i} \wedge \mathrm{Trl}(\mathsf{cmd_i}) > \qquad (*)$$

gets translated to

$$\mathcal{WF}(\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|} \mathsf{guard_i}, (\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|} \mathsf{guard_i}) \wedge \mathrm{Trl}(\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|} \mathsf{guard_i} \wedge \mathrm{Trl}(\mathsf{cmd_i}))).$$

This is equivalent to

$$\mathcal{WF}(\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|}\mathsf{guard_i},(\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|}\mathsf{guard_i})\ \wedge\ \bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|}\mathsf{guard_i}\wedge\mathrm{Trl}(\mathsf{cmd_i}))$$

since $\mathrm{Trl}(\mathsf{p})=\mathsf{p}$ for transition predicates p. Furthermore, $\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|}\mathsf{guard_i}\wedge\mathrm{Trl}(\mathsf{cmd_i})$ implies $\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|}\mathsf{guard_i}$. Therefore, (*) is equivalent to local progress:

$$\mathcal{WF}(\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|}\mathsf{guard_i},\ \bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|}\mathsf{guard_i}\wedge\mathrm{Trl}(\mathsf{cmd_i})).$$

It remains to be shown that [F1], [F2] and [F3] hold for local progress:

- [F1] By definition, $\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|}\mathsf{guard_i}$ is a state predicate such that
$$\mathsf{FFV}(\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|}\mathsf{guard_i})\ \subseteq\ \mathrm{Ctr_V}(\mathsf{block})$$

- [F2] By definition, $\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|}\mathsf{guard_i}\wedge\mathrm{Trl}(\mathsf{cmd_i})$ is a command such that
$$\mathrm{Ctr}(\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|}\mathsf{guard_i}\wedge\mathrm{Trl}(\mathsf{cmd_i}))\ \subseteq\ \mathrm{Ctr}(\mathsf{block})$$

- [F3] $[\![\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|}\mathsf{guard_i}\ \Rightarrow\ \exists_{\mathrm{Ctr'}(\mathsf{block})}\mathrm{Trl}(\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|}\mathsf{guard_i}\wedge\mathrm{Trl}(\mathsf{cmd_i}))\wedge\delta_{\mathsf{ctr}}(\mathsf{block})\,]\!]$ holds:

  Let ξ be a model of $\bigvee_{|tt|<i\leq|tt|+|gt|}\mathsf{guard}_i$. It has to been shown that

  $$[\![\,\exists_{\mathrm{Ctr'}(\mathsf{block})}\mathrm{Trl}(\bigvee_{|\mathsf{tt}|<\mathsf{i}\leq|\mathsf{tt}|+|\mathsf{gt}|}\mathsf{guard_i}\wedge\mathrm{Trl}(\mathsf{cmd_i}))\wedge\delta_{\mathsf{ctr}}(\mathsf{block})\,]\!]^{(\xi,\alpha,\xi')}\qquad(**)$$

  holds for arbitrary α and ξ'.

  ξ is model of guard_i for some i. Due to [GT], there are valuations $\hat{\alpha}$ and $\hat{\xi}'$ such that $[\![\,\mathsf{guard}_i\wedge\mathrm{Trl}(\mathsf{cmd}_i)\,]\!]^{(\xi,\hat{\alpha},\hat{\xi}')}$.

  Then define $\tilde{\alpha}(A)\stackrel{\mathrm{def}}{=}\begin{cases}\hat{\alpha}(A) & \text{, if } A\in\mathrm{Ctr}(\mathsf{cmd}_i)\\ \bot & \text{, else}\end{cases}$

  and $\tilde{\xi}'(x')\stackrel{\mathrm{def}}{=}\begin{cases}\hat{\xi}'(x') & \text{, if } x\in\mathrm{Ctr}(\mathsf{cmd}_i)\\ \xi(x) & \text{, else}\end{cases}$.

  Obviously, $(\xi,\tilde{\alpha},\tilde{\xi}')$ is a model of both $\bigvee_{|tt|<i\leq|tt|+|gt|}guard_i\wedge\mathrm{Trl}(cmd_i)$ and $\delta_{ctr}(\mathsf{block})$. Because all actions and primed variables that occur free in $\mathrm{Trl}(\bigvee_{|tt|<i\leq|tt|+|gt|}guard_i\wedge\mathrm{Trl}(cmd_i))\wedge\delta_{ctr}(\mathsf{block})$ are elements of $\mathrm{Ctr'}(\mathsf{block})$, (**) holds for arbitrary α and ξ'.

■

Lemma 5.7.16 $\Sigma^a(\mathsf{block})$ *is well-defined.*

Proof:

1. $J \subseteq W$ by definition,

2. $\emptyset \neq J$:
   It has to be shown that there is some $p \in W$ and some valuation $\beta : \mathrm{Ctr}_V(\mathsf{block}) \to \mathcal{U}$ such that $[\![\mathsf{p} \wedge \mathsf{init}]\!]^\beta$. Assume not. Then $[\![(\bigvee_{\mathsf{p} \in \mathsf{W}} \mathsf{p}) \wedge \mathsf{init}]\!]^\beta$ does not hold either (for any β).
   Due to [aBTS3], it follows that $[\![\mathsf{init}]\!]^\beta$ does not hold for any β. This contradicts [Init2].

3. $\mathcal{B}$ is a complete, atomic Boolean algebra by definition;

4. $N : W \times W \to \mathcal{B}$ by definition,

5. $st \cap N(p,p) \neq \emptyset$ for all $p \in W$: Because of [aBTS1], there is a valuation ξ of the variables in $\mathrm{Ctr}_V(\mathsf{block})$ such that $[\![p]\!]^\xi$. Due to lemma 5.7.6 it is sufficient to define $\mu(x) \stackrel{\text{def}}{=} \begin{cases} \xi(x) & \text{, for } x \in \mathrm{Ctr}_V(\mathsf{block}) \\ \text{arb.} & \text{, for } x \notin \mathrm{Ctr}_V(\mathsf{block}) \end{cases}$. Then $(\mu, \tau, \mu) \in st \cap N(v,v)$.

6. $\emptyset \neq L(e) \cap N(e)$ for all $e = (p,q) \in E$:
   If $e \in E$ then there exists some μ such that $[\![p \wedge q' \wedge \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block})]\!]^\mu$. Then define $\nu(a) = \begin{cases} \bot & \text{, } a \in \mathsf{Acts} - \mathrm{Ctr}(\mathsf{block}) \\ \mu(a) & \text{, else} \end{cases}$.

   The actions from $\mathsf{Acts} - \mathrm{Ctr}(\mathsf{block})$ do not occur in $p \wedge q' \wedge \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block})$. Therefore, $[\![p \wedge q' \wedge \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block})]\!]^\nu$.
   On the other side, besides $[\![p \wedge q' \wedge \delta_{ctr}(\mathsf{block})]\!]^\nu$ also $[\![\delta_{env}(\mathsf{block})]\!]^\nu$ holds due to Corollary 5.6.3, because all actions occurring in events are elements from $\mathsf{Acts} - \mathrm{Ctr}(\mathsf{block})$ (which is due to condition [Ev] in Definition 5.7.1). That is, $\nu \in N(e)$.

■

Theorem 5.7.17 (Relationship between models)

$\mathbf{Traces}(\Sigma(\mathsf{block})) \subseteq \mathbf{Traces}(\Sigma^a(\mathsf{block}))$

Proof:

Let $\tau = \xi_0 \xrightarrow{\alpha_0} \xi_1 \xrightarrow{\alpha_1} \xi_2 \cdots$. $\mathbf{Traces}(\Sigma(\mathsf{block})) \subseteq \mathbf{Traces}(\Sigma^a(\mathsf{block}))$ may be established by proving

There is a sequence $(v_0, v_1, \dots) \in V^\omega$ such that		There is a sequence $(w_0, w_1, \dots) \in W^\omega$ such that
[V1] $v_0 \in I$		[W1] $w_0 \in J$
[V2] $(\xi_i, \alpha_i, \xi_{i+1}) \in M(v_i, v_{i+1})$ for all i		[W2] $(\xi_i, \alpha_i, \xi_{i+1}) \in N(w_i, w_{i+1})$ for all i
[V3a] continuously $v_i \in G$ implies infinitely often $(v_i, v_{i+1}) \in E$ and $(\xi_i, \alpha_i, \xi_{i+1}) \in L(v_i, v_{i+1})$ for all $\mathsf{WF}(G, E, L) \in \mathcal{F}$	**implies**	[W3a] continuously $w_i \in G^a$ implies infinitely often $(w_i, w_{i+1}) \in E^a$ and $(\xi_i, \alpha_i, \xi_{i+1}) \in L^a(w_i, w_{i+1})$ for all $\mathsf{WF}(G^a, E^a, L^a) \in \mathcal{G}$
[V3b] infinitely often $v_i \in G$ implies infinitely often $(v_i, v_{i+1}) \in E$ and $(\xi_i, \alpha_i, \xi_{i+1}) \in L(v_i, v_{i+1})$ for all $\mathsf{SF}(G, E, L) \in \mathcal{F}$		[W3b] infinitely often $w_i \in G^a$ implies infinitely often $(w_i, w_{i+1}) \in E^a$ and $(\xi_i, \alpha_i, \xi_{i+1}) \in L^a(w_i, w_{i+1})$ for all $\mathsf{SF}(G^a, E^a, L^a) \in \mathcal{G}$

Firstly, let w_i be the (uniquely determined) $w \in W$ such that $[\![w_i]\!]^{\xi_i}$.

[V1] $\Rightarrow$ [W1]:

If $v_0 \in I$, then by definition $[\![\mathsf{init}]\!]^{v_0}$ and (due to [V2]) also $[\![\mathsf{init}]\!]^{\xi_0}$. Then $[\![w_0 \wedge \mathsf{init}]\!]^{\xi_0}$ by definition of w_i. Thus, $[\![\exists_{\mathrm{Ctr}_V(\mathsf{block})} (w_0 \wedge \mathsf{init})]\!]$, i.e. $w_0 \in J$.

[V2] $\Rightarrow$ [W2]:

Let $(\xi_i, \alpha_i, \xi_{i+1}) \in M(v_i, v_{i+1})$. Then especially, $[\![\delta(\mathsf{block})]\!]^{(\xi_i, \alpha_i, \xi_{i+1})}$. Thus $[\![\delta(\mathsf{block}) \wedge w_i \wedge {w_{i+1}}']\!]^{(\xi_i, \alpha_i, \xi_{i+1})}$ by definition of w_i, i.e. $(\xi_i, \alpha_i, \xi_{i+1}) \in N(w_i, w_{i+1})$.

[V3a] $\Rightarrow$ [W3a]:

Suppose continuously $w_i \in G^a$ for some explicit fairness condition $\mathsf{WF}(G^a, E^a, L^a) \in \mathcal{G}$. I.e. $[\![w_i \Rightarrow \mathsf{fg}]\!]$ and there is some μ such that $[\![w_i \wedge {w_{i+1}}' \wedge \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block})]\!]^{\mu}$. Especially, $[\![w_i \Rightarrow \mathsf{fg}]\!]^{(\xi_i, \alpha_i, \xi_{i+1})}$ and by definition of w_i also $[\![\mathsf{fg}]\!]^{(\xi_i, \alpha_i, \xi_{i+1})}$. Because of [V2], $\xi_i(x) = v_i(x)$ for all $x \in \mathsf{FFV}(\mathsf{fg}) \subseteq \mathrm{Ctr}_V(\mathsf{block})$. Thus $[\![\mathsf{fg}]\!]^{v_i}$ and by [F3] it follows that there is some η_i such that $[\![\mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block})]\!]^{\eta_i}$ and $\eta_i(x) = v_i(x)$ for all $x \in \mathrm{Ctr}_V(\mathsf{block})$. Therefore, (continuously) $v_i \in G$ and thus always eventually $(v_i, v_{i+1}) \in E$ and $(\xi_i, \alpha_i, \xi_{i+1}) \in L(v_i, v_{i+1})$. This implies (always eventually) $[\![\mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block})]\!]^{(\xi_i, \alpha_i, \xi_{i+1})}$. By definition of w_i it follows that (always eventually) $[\![w_i \wedge {w_{i+1}}' \wedge \mathsf{fg} \wedge \mathrm{Trl}(\mathsf{fcmd}) \wedge \delta_{ctr}(\mathsf{block})]\!]^{(\xi_i, \alpha_i, \xi_{i+1})}$. Thus always eventually $(w_i, w_{i+1}) \in E^a$ and $(\xi_i, \alpha_i, \xi_{i+1}) \in L^a(w_i, w_{i+1})$.

[V3b] $\Rightarrow$ [W3b]:

Analog (simply replace "continuously" by "always eventually").

■

Bibliography

[AG94] Robert Allen and David Garlan. Formal Connectors. Technical Report CMU-CS-94-115, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, March 1994.

[AL88] M. Abadi and L. Lamport. The existence of refinement mappings. In Computer Society Press, editor, *Proc. of the 3rd Annual IEEE Symp. on Logic in Computer Science*, pages 165–175, July 1988. Washington D.C.

[AL92] M. Abadi and L. Lamport. An Old-Fashioned Recipe for Real Time. Technical Report 91, DEC Systems Research Center, 1992.

[AL93] Martin Abadi and Leslie Lamport. Conjoining Specifications. Technical Report 118, Systems Rsearch Center, Digital, Palo Alto, California, December 1993.

[And86] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, New York and London, 1986.

[BA93] M. Ben-Ari. *Mathematical Logic in Computer Science.* Prentice-Hall, Hemel, Hampstead, 1993.

[BAPM83] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, pages 207–226, 1983.

[Bar97] D. Barnard. *Temporal Language of Transitions and Client-Server-Systems.* PhD thesis, Faculty of Informatics, Technical University of Munich, D-80290 Munich, Germany, 1997.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time programming. In *Proc. of IEEE*, volume 79(9), pages 1270–1282, 1991.

[BC95] Dieter Barnard and Simon Crosby. The Specification and Verification of an ATM Signalling Protocol. In *Proc. of 15th IFIP PSTV'95*, Warsaw, June 1995.

[BCG88] M.C. Browne, E.M. Clarke, and O. Grümberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.

[Ber93] Daniel M. Berry. Formal specification and verification of concurrent programs. Technical Report SEI-CM-27-1.0, School of Computer Science, CMU, Pittsburgh, February 1993.

[Bro96] Manfred Broy. A Functional Solution to the RPC-Memory Specification Problem. Submitted as a Final Solution to Dagstuhl Seminar of Broy/Lamport, 1994, Faculty of Informatics, Technical University of Munich, D-80290 Munich, Germany, 1996.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, August 1986.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[Bus95] H. Busch. A Practical Method for Reasoning About Distributed Systems in a Theorem Prover. In *Higher Order Logic Theorem Proving and its Applications - 8th International Workshop, Aspen Grove, UT, USA, Proceedings*, pages 106–121. Springer-Verlag, LNCS 971, September 1995.

[Bus96] H. Busch. Proving liveness of fair transition systems. In J. v. Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOL'96*, volume 1125 of *LNCS*, pages 77–92. Springer-Verlag, August 1996.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Workshop on Logic of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.

[CES83] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Proceedings 10th ACM Symposium on Principles of Programming Languages*, pages 117–126, 1983.

[CGS91] C. Courcoubetis, S. Graf, and J. Sifakis. An Algebra of Boolean Processes. In *Proc. of CAV'91*, pages 454–465, 1991.

[CH93] Jorge Cuéllar and Martin Huber. Property Preserving Datarefinement. Internal report, Siemens Corporate Research and Development, ZFE T SE 1, D-81730 Munich, Germany, 1993.

[CH95] Jorge Cuéllar and Martin Huber. The FZI Production Cell Case Study: A distributed solution using TLT. In *Formal Development of Reactive Systems: Case Study Production Cell*, volume 891 of *LNCS*. Springer-Verlag, 1995.

[CHB96a] Jorge Cuéllar, Martin Huber, and Dieter Barnard. A Solution relying on the Model Checking of Boolean Transition Systems. In *The RPC-Memory Specification Problem*, LNCS 1169, 1996.

[CHB96b] Jorge Cuéllar, Martin Huber, and Dieter Barnard. Rapid Protyping for an Assertional Specification Language. *TACAS'96, LNCS 1055*, March 1996.

[CK93] Pierre Collette and Edgar Knapp. A Compositional Proof System for UNITY Based on Rely/Guarantee Conditions. In *Proc. of IFIP Working Conf. on Programming Concepts, Methods and Calculi*. IFIP, 1993.

[CM88] K. Mani Chandy and J. Misra. *Parallel Program Design - A Foundation*. Addison-Wesley, Reading, Massachusetts, 1988.

[Col94] Pierre Collette. An explanatory presentation of composition rules for assumption-commitment specifications. *Information Processing Letters*, 50:31–35, 1994.

[CW96] Jorge Cuéllar and Isolde Wildgruber. The Steam Boiler Problem - A TLT Solution (Presented at a Dagstuhl Seminar). In *Proc. of a Dagstuhl Seminar*, 1996.

[CWB94] J. R. Cuéllar, I. Wildgruber, and D. Barnard. Combining the Design of Industrial Systems with Effective Verification Techniques. In M. Naftalin, T. Denvir, and M. Betran, editors, *Proc. of FME'94*, volume 873 of *LNCS*, pages 639–658, Barcelona, Spain, October 1994. Springer-Verlag.

[DF95] Jürgen Dingel and Thomas Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Proc. of CAV'95*, LNCS 939, Liege, Belgium, June 1995. Spinger-Verlag.

[EH86] E.A. Emerson and J.Y. Halpern. 'sometimes' and 'not never' revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[EL85] E. Allen Emerson and C.L. Lei. Modalities for Model Checking: Branching Time Strikes Back. In *Proc. of 12th Annual ACM Sympo. on POPL*, pages 84–96, New Orleans, Louisiana, January 1985. ACM.

[Eme90] E. Allen Emerson. *Temporal and Modal Logic*, chapter 16, pages 996–1072. Handbook of Theoretical Computer Science. Elsevier Science Publishers B.V., edited by j. van leeuwen edition, 1990.

[Fit90] Fitting. *First Order Logic and Automated Theorem Proving.* Springer, New York, 1990.

[Flo67] R. W. Floyd. Assigning Meanings to Programs. In *Proc. of the Symposium on Applied Mathematics*, pages 19–32, American Mathematical Society, 1967.

[Fra86] Nissim Francez. *Fairness.* Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 1986.

[FSS+94] T. Filkorn, H.A. Schneider, A. Scholz, A. Strasser, and P. Warkentin. SVE User's Guide. Technical report, Siemens AG, ZFE T SE 1, D-81730 München, Germany, 1994.

[GK80] B. Gopinath and R.P. Kurshan. The selection/resolution model for coordinating concurrent processes. Technical report, Mathematical Sciences Research Center, AT&T Bell Laboratories, Murray Hill, NJ, 1980.

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *USING MPI - Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, MA, 1994.

[Gri81] David GriesFitting. *The Science of Programming.* Springer, New York, 1981.

[Gur91] Y. Gurevich. Evolving Algebras: A Tutorial Introduction. In *Bulletin of the EATCS*, volume 43, pages 264–284. EATCS, 1991.

[Hal74] Paul R. Halmos. *Lectures on Boolean algebras.* Springer-Verlag, 1974.

[Hal93] N. Halbwachs. *Synchronous Programming of reactive systems.* Kluwer, 1993.

[HL95] M. Hennessey and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.

[Hoa69] C.A.R. Hoare. An axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, London, 1985.

[Hub93] M. Huber. Verfeinerung in TLT. Master's thesis, Universität Karlsruhe, 1993.

[Kal96] Markus Kaltenbach. *Model Checking for UNITY.* PhD thesis, University of Texas at Austin, 1996.

[Kur94] R.P. Kurshan. *Computer Aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[Lam80] Leslie Lamport. 'sometimes' is sometimes 'not never'. In *Proc. of 7th Annual ACM Symp. on POPL*, Las Vegas, January 1980. ACM.

[Lam91] Leslie Lamport. The Temporal Logic of Actions. Technical Report 79, Digital Systems Research Center, Palo Alto, California, December 1991.

[Lam94a] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam94b] Leslie Lamport. TLA+. Technical Report (preliminary), DEC Systems Research Center, August 1994.

[LL95] Claus Lewerentz and Thomas Lindner. *Formal Development of Reactive Systems: Case Study Production Cell*, volume 891 of *Lecture Notes in Computer Science.* Springer-Verlag, January 1995.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierachical Correctness Proofs for Distributed Algorithms. In *Proc. of 6th Symp. on Principles of Distr. Comp.*, pages 137–151. ACM, 1987.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/putput automata. *CWI-Quarterly*, 2(3), 1989.

[Mer94] Stephan Merz. Explaining Composition. Technical report, Institute of Informatics, Technical University of Munich, Munich, Germany, July 1994.

[Mer95] Stephan Merz. TLA-Semantik für U-TLT Programme. Internal paper, Faculty of Informatics, Technical University of Munich, D-80290 Munich, March 1995.

[Mer96] Stephan Merz. From TLT modules to stream processing functions. Internal paper, Faculty of Informatics, Technical University of Munich, D-80290 Munich, March 1996.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science 92. Springer-Verlag, Berlin, 1980.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, London, 1989.

[Mis96] Jayadev Misra. A discipline of multiprogramming - preliminary draft. Technical report, University of Austin, 1996.

[MP81] Zohar Manna and Amir Pnueli. Verification of concurrent programs, part i: The temporal framework. Technical Report STAN-CS-81-836, Department of Computer Science, Stanford University, July 1981.

[MP91] Z. Manna and A. Pneuli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New York, 1991.

[MP96] Z. Manna and A. Pneuli. *Temporal Verification of Reactive Systems - Safety.* Springer-Verlag, New York, 1996.

[Rat97a] Rational. The unified modeling language v1.1 documentation set. Technical report, Rational Software Corporation, Monterey California, 1997.

[Rat97b] Rational. The unified modeling language v1.1 semantics. Technical report, Rational Software Corporation, Monterey California, 1997.

[RC90] Gruia-Catalin Roman and H. Conrad Cunningham. A unity-style programming logic for shared dataspace programs. *IEEE Transactions on Parallel and Distributed Systems*, 1(3), 1990.

[RCM+95] Hans Rischel, Jorge Cuéllar, Simon Mørk, Anders P. Ravn, and Isolde Wildgruber. Development of Safety-Critical Real-Time Systems. In *Proceedings of the SOFSEM '95, M.Bartosek, J.Staudek, J.Wiedermann (Eds)*, volume 1012 of *LNCS*, November 1995.

[Ric89] Michael M. Richter. *Prinzipien der Künstlichen Intelligenz.* Teubner, Stuttgart, 1989.

[San91] B. A. Sanders. Eliminating the Substitution Axiom from UNITY Logic. *Formal Aspects of Computing*, 3:189–205, 1991.

[Sch89] U. Schöning. *Logik für Informatiker.* BI-Wissenschaftsverlag, 1989.

[Sti92] Colin Stirling. Modal and temporal logics. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[tcN92] IEC technical committee No. 65. International Standard IEC 1131. Technical report, International Electrotechnical Commission, Denmark, 1992.

[UHK94] Rob Udink, Ted Herman, and Joost Kok. Progress for Lokal Variables in UNITY. In *Proc. of IFIP Working Conf. on Programming Concepts, Methods and Calculi.* IFIP, 1994.

[UK92] R.T. Udink and J.N. Kok. On the relation between unity properties and sequences of states. In *Proc. of REX'92 Workshop*, volume Unknown of *LNCS*, pages 594–608. Springer-Verlag, 1992.

Keywords and Symbols

Consistency Conditions

Index

LEBENSLAUF

2.10.1967	geboren in Karlsruhe
1978 – 1987	Gymnasium in Oberkirch, Abitur
1987 – 1988	Grundwehrdienst in Sigmaringen
1988 – 1993	Studium der Informatik mit Nebenfach Mathematik an der Universität Karlsruhe. Diplom im Herbst 1993
1989	Aufnahme in die Studienstiftung des deutschen Volkes
1990 – 1991	Hilfswissenschaftler an der Fakultät für Mathematik
1993 – 1996	Forschungsaufenthalt in der Zentralabteilung Technik der Siemens AG in München.
seit 1996	Mitarbeiter in der Zentralabteilung Technik der Siemens AG in Erlangen.