Interactive Verification of Synchronous Systems

Manuel Gesell

vom Fachbereich Informatik der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

Datum der wissenschaftlichen Aussprache	07. November 2014
Dekan	Prof. Dr. Klaus Schneider
Vorsitzender der Promotionskommission	Prof. Dr. Markus Nebel
Berichterstatter	Prof. Dr. Klaus Schneider Prof. Dr. Rolf Drechsler

D 386

Danksagung

Mein Dank gilt allen, die auf ihre Weise zum Gelingen meiner Promotion beigetragen haben. Im Besonderen möchte ich mich bei meinem Doktorvater Prof. Dr. Klaus Schneider bedanken, der mir immer beratend zur Seite stand und mich unterstütze. Ich werde die interesannten Diskussionen und die gemeinsame Arbeit sehr vermissen, aber auch gerne darauf zurückblicken.

Für die Zweitbegutachtung möchte ich mich bei Prof. Dr. Rolf Drechsler bedanken. Für die Leitung der Promotionskommission bedanke ich mich bei Prof. Dr. Markus Nebel, der sehr kurzfristig dieses Amt übernommen hat.

Darüberhinaus bedanke ich mich bei den Kollegen in meiner Arbeitsgruppe, die mir in Gesprächen einige neue Denkanstöße gaben. Besonders möchte ich dabei Dr. Andreas Morgenstern hervorheben, mit dem ich mehrere Publikationen erstellen durfte.

Mein größter Dank gilt jedoch meiner Frau Johanna; ohne ihre Liebe und Unterstützung hätte ich weder die Zeit noch die Kraft gehabt, diese Arbeit zu verfassen. Meinen Töchtern Zoey und Yuna danke ich dafür, dass sie es ihrer Mutter vor allem in der Endphase dieser Arbeit nicht allzu schwer gemacht haben.

Mannheim, 07. November 2014 *Manuel Gesell*

Zusammenfassung

Eingebette Systeme reichen von sehr einfachen Systemen zu sehr komplexen Steuerungen, und haben meist extreme Echtzeit- und Planungsanforderungen. Viele Eingebettete Systeme sind reaktive Systeme, die auf Ereignisse der Umgebung reagieren müssen und dabei bestimmte zeitliche Vorgaben erfüllen müssen. Die Ausführung solcher Systeme wird normalerweise in Reaktionsschritte unterteilt, wobei in jedem Schritt das System seine Eingaben liest und auf diese mit den entsprechenden Ausgaben reagiert.

Das synchrone Berechnungsmodel (MoC) ist geeignet solche Systeme zu modellieren, da es dem Paradigma der perfekten Synchronie folgt, welches die Natur der reaktiven Systeme wiederspiegelt. Ein weiterer Vorteil ist das Vorhandensein von formalen Verifikationsmethoden, wie Modelprüfung. Die Anwendbarkeit dieser Methoden basiert auf der deterministischen Ausführung solcher Systeme, aufgrund der formalen Semantik. Durch die steigende Komplexität von Eingebetteten Systemen muss der natürliche Nachteil der Modellprüfung, die Zustandsraumexplosion, ausgeglichen werden um akzeptable Laufzeiten des Verifikationsverfahren sicher zu stellen. Dafür werden meist bestehende Techniken wiederverwendet. Vor allem Methoden, die den Zustandsraum durch z.B. Zerlegung verkleinern und damit dem Nachteil der Modellprüfung entgegenwirken sind wichtig. Die Definition solcher Dekompositionstechniken ist für synchrone Sprachen, durch deren Parallelität nicht einfach.

Inspiriert durch die Erfolge im Gebiet der Desynchronisation von synchronen Systemen, welche diese in asynchrone Systeme transformieren, wird diese Arbeit die Möglichkeit untersuchen, ob ähnliche Techniken auch für die Verifikation verwendet werden können. Dabei werden Techniken betrachtet, die ein anderes Berechnungsmodell zu Grunde legen und eine mögliche Verwendung für die Verifikation synchroner Systeme untersucht. Dabei wird der Schwerpunkt auf die interaktive Verifikation synchroner Systeme, basierend auf dem Hoare-Kalkül, einer grundlegenden Verifikationstechnik sequentieller Programme gelegt. Durch die verschiedenen zugrundeliegenden Berechnungsmodelle muss eine vielzahl an Problemen gelöst werden. Besonders problematisch ist die Möglichkeit, dass verschiedene Programmteile zum gleichen Zeitpunkt aktiv sein und sich gegenseitig beeinflussen können. Im Gegensatz zum sequentiellen Fall, bei dem man einzelne Anweisungen betrachtet, bedeutet eine Dekomposition synchroner Programme, dass man eine symbolische Ausführung von mehreren parallelen Programteilen betrachten muss. Zum Lösen dieses Problems werden verschiedene Ansätze gezeigt.

Außerdem wird gezeigt wie ein synchrones System durch andere Berechnungsmodelle zum Zwecke der Verifikation dargestellt werden kann und welchen Einfluß dies auf dessen Verifikation hat.

Die Realisierbarkeit der vorgestellten Ansätze wird durch die Integration mit den vorhandenen Modellprüfungsmethoden durch die Implementierung eines Prototyps gezeigt.

Abstract

Embedded systems, ranging from very simple systems up to complex controllers, may nowadays have quite challenging real-time requirements. Many embedded systems are reactive systems that have to respond to environmental events and have to guarantee certain real-time constrain. Their execution is usually divided into reaction steps, where in each step, the system reads inputs from the environment and reacts to these by computing corresponding outputs.

The synchronous *Model of Computation* (MoC) has proven to be well-suited for the development of reactive real-time embedded systems whose paradigm directly reflects the reactive nature of the systems it describes. Another advantage is the availability of formal verification by model checking as a result of the deterministic execution based on a formal semantics. Nevertheless, the increasing complexity of embedded systems requires to compensate the natural disadvantages of model checking that suffers from the well-known *state-space explosion problem*. It is therefore natural to try to integrate other verification methods with the already established techniques. Hence, improvements to encounter these problems are required, e.g., appropriate decomposition techniques, which encounter the disadvantages of the model checking approach naturally. But defining decomposition techniques for synchronous language is a difficult task, as a result of the inherent parallelism emerging from the synchronous broadcast communication.

Inspired by the progress in the field of desynchronization of synchronous systems by representing them in other MoCs, this work will investigate the possibility of adapting and use methods and tools designed for other MoC for the verification of systems represented in the synchronous MoC. Therefore, this work introduces the interactive verification of synchronous systems based on the basic foundation of formal verification for sequential programs – the Hoare calculus. Due to the *different models of computation* several problems have to be solved. In particular due to the large amount of concurrency, several parts of the program are active at the same point of time. In contrast to sequential programs, a decomposition in the Hoare-logic style that is in some sense a symbolic execution from one control flow location to another one requires the consideration of several flows here. Therefore, different approaches for the interactive verification of synchronous systems are presented.

Additionally, the representation of synchronous systems by other MoCs and the influence of the representation on the verification task by differently embedding synchronous system in a single verification tool are elaborated.

The feasibility is shown by integration of the presented approach with the established model checking methods by implementing the AIFProver on top of the Averest system.

Contents

Chapter 1

Introduction

1.1 Motivation

Embedded systems, ranging from very simple systems up to complex controllers, may nowadays have quite challenging real-time requirements. Many embedded systems are reactive systems [HaPn85] that have to respond to environmental events and have to guarantee certain real-time constrain. Their execution is usually divided into reaction steps, where in each step, the system reads inputs from the environment and reacts to these by computing corresponding outputs. The synchronous *Model of Computation* (MoC) [Halb93, BCEH03] has proven to be well-suited for the development of these systems [HaPn85] whose paradigm directly reflects the reactive nature of the systems it describes. The predictable and deterministic nature of this MoC allows one to determine the worst case execution time (WCET), which is an important property for these systems. Languages like Esterel [BeGo92] and Quartz [Schn09] that are control-flow based languages which implement the synchronous MoC provide many convenient statements to control the execution. The explicit notion of (logical) time also requires different kinds of assignments to enable communication within a step (immediate assignments) and between successive steps (delayed assignments). Additionally, they support the development of embedded systems consisting of application-specific hardware *and* software since compilers can generate hardware as well as software from the same synchronous program. For safety-critical applications, an important advantage of synchronous programs is that formal verification can be directly applied due to the availability of precisely defined formal semantics for all synchronous programming languages. This allows one to translate programs into e.g. state transition systems and to make use of model checking [GrVe08, ClGP99] to verify the system's behavior. Such procedures are even integrated within the compilers, e.g. to check for instantaneous loops, to guarantee the absence of write conflicts and runtime errors, to analyze causality problems, and to solve many other problems that might appear during compilation.

However, it is well-known that model checking suffers from its enormous complexity meaning that only systems of a moderate size can be verified this way [BCMD92]. It is therefore natural to try to integrate other verification methods with the already established techniques[1]. Hence, improvements to encounter these problems are required, e.g., appropriate decomposition techniques, which encounter the disadvantages of the model checking approach naturally. But defining decomposition techniques for synchronous languages is a difficult task due to the inherent parallelism emerging from the broadcast communication.

Inspired by desynchronization techniques [CaMS01, Gira05, CaCo09, PoCB04] that allow to run a synchronous program separated in multiple threads instead of a single one to be executed on multi-core machines or distributed systems by a MoC transformation, this work will use similar transformations for verification. The integration of existing verification techniques developed for other MoCs might be beneficial for the synchronous MoC. Especially interactive verification methods based on theorem provers are an interesting alternative for verification of large systems at different abstraction levels. Interactive verification has many advantages: It allows one to decompose a proof task to several simpler tasks, to abstract from the size of data structures as well as the data types itself, and to exploit the user's knowledge of the program to e.g. encounter the state-space-explosion problem of the established model checking methods for synchronous languages. However, state-of-the-art theorem provers were designed for general proof problems, so that their use for the interactive verification of synchronous/reactive systems is often too inconvenient.

In particular, the Hoare calculus [Floy67, Hoar69, Grie81, Apt81, Cous90] a basic foundation for the formal verification of sequential programs, has not been considered for synchronous systems before. This is surprising since it was adapted to different concurrent languages [OwGr76a, OwGr76b, ApOl97, RBHH01] and even though the operational semantics of synchronous systems seems to be compatible with the Hoare rules that define the meaning of (sequential) programs. For example, the semantics of the imperative synchronous language Quartz is defined by Plotkin's *Structural Operational Semantics (SOS)* rules [Plot81, Schn09]. Hence, to establish a Hoare calculus for Quartz, it seems that these two sets of rules *only* need to be combined.

Another advantage of changing the MoC is that it allows one to use many successful rewrite-based (verification) tools, theorem provers and languages, like SAL [MORR04], Rodin [Abri10], BlueSpec [Arvi03], CAOS [SiSh07], Murphi [Dill96], UNITY [ChMi89], VAMPIRE [KoVo13], Giotto[HeHK01] and many others that are based on concurrent/asynchronous MoCs similar to Dijkstra's guarded commands [Dijk75] and currently not usable to analyze systems of the synchronous MoC.

In general, guarded actions are well suited to describe systems with different models of computations (MoCs). Different MoCs are thereby obtained by different ways to select actions for execution: In particular, in synchronous models, one executes *all* enabled actions synchronously in the same environment, in interleaved models, only a single (enabled) guarded action is chosen (in a non-deterministic way) for execution, and in asynchronous models, some non-empty *subset* of the enabled actions is (non-deterministically) executed. It seems to be impossible to use tools for one of these MoCs for guarded actions that are based on another MoC. This holds especially for the synchronous MoC, because the execution of all enabled guarded actions in a step follows implicitly a causal ordering to define a deterministic

[1] It is interesting to note that some recent approaches to software verification do the converse, i.e. try to employ model checking for the verification of sequential software programs [Gode05, CCGO04, BaPR02].

result, hence confluence must be ensured to guarantee deterministic results for the interleaved execution.

In summary it can be said, that the verification of synchronous systems might benefit from tools and verification techniques for other MoCs.

1.2 Content of the Thesis

This thesis shows that the synchronous MoC does not prohibit the application or adaption of verification techniques and tools developed for other MoCs, and furthermore, that even the combination of those methods with model checking, which is the dominant verification technique for synchronous languages, is possible. The enormous number of existing MoCs and verification techniques therefore makes it impossible to discuss them all, hence only some successful approaches are chosen and the work concentrates on them.

As a result of the already mentioned arguments, this work will present a feasibility study to enable interactive verification for synchronous languages. The main goal is the evaluation of possible ways to interactively verify synchronous languages by adapting existing techniques for other MoCs (e.g. sequential languages) such that the existing model checking methods for synchronous languages are supported during the verification task by e.g. suitable decomposition rules. Therefore, the following problems have to be solved:

- Evaluation of methods for the interactive decomposition of synchronous languages, e.g. Hoare-calculus [Hoar69], temporal logic decomposition methods, e.g. compositional model checking [McQS00, BeCC98] and de Roever's concurrency verification [RBHH01], and module checking coined by Kupferman and Vardi [KuVW01].
- Enabling the re-usability of existing decision procedures (developed for other MoCs) for synchronous languages

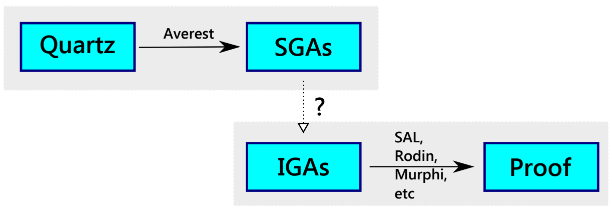
This work will start with enabling the use of the Hoare calculus for Quartz. Since Quartz programs can be translated to sequential programs by the Averest system, it is in principle possible to apply the classic Hoare calculus to the result of these translations. However, the translations destroy most of the original program's syntax: Usually, compilers generate sequential programs that consist of only a single loop whose body contains the code for all macro steps [PoEB07]. Thus, most of the syntax of the synchronous program is lost, and since the Hoare calculus requires additional information from the user and is directed by the syntax, it does not really make sense to apply the Hoare calculus after such a translation. Another way to make use of this calculus is the definition of rules for the additional used synchronous statements based on the available SOS rules. This thesis sketches this approach and shows that a huge set of complicated rules are required, because besides defining the semantics of each statement, these rules need to handle problems specific to synchronous languages, e.g. schizophrenia, causality, as well as micro step and macro step behaviors. Furthermore, they have to solve the issue that the compositional approach of Hoare calculus is difficult to implement, because in a Quartz program several control flows may be active. To circumvent most of the encountered issues, a normal form for the Quartz source code will be defined that aggregates all behavioral changes (micro step assignments) into a single synchronous tuple assignment (STA) such that the reasoning about the behavior is simplified by collecting the programs behavior at a single program part (the STA). This normal form does not contain parallel statements, and has therefore properties similar to a sequential program. Hence, it allows the compositional reasoning with a small set of proof rules and can be seen as a pseudo-transformation to the sequential MoC. This way, a convenient use of interactive verification based on an adapted Hoare calculus is possible. For this purpose a program transformation at the source code level is required. Unfortunately, a proof will show that it is in general impossible to define a syntax-preserving transformation without adding additional variables. Nevertheless, two incomplete transformations covering most Quartz programs are presented.

After this unsatisfactory result, this thesis will present another way to interactively verify synchronous systems that is applicable to all Quartz programs. Therefore, rules operating on synchronous guarded actions (SGAs), which are generated by compiling a Quartz program [GeSc13a, BrSc09, Schn09], are defined. This approach has several advantages: the transformations implemented in the compiler are reusable without additional effort, and using guarded actions instead of the original source code allows more flexible decompositions of proof goals: There is no need to follow the syntax of the program. Due to a back-annotation via control-flow locations, there is still a direct relation between the source code and the compiled guarded actions. Hence, the user works still on the readable source code, while the rules operate on the compiled code.

Then, existing decomposition/verification techniques based on temporal logic [RBHH01, McQS00, RBHH01, MaPn95b, MaPn92] are evaluated to combine the approach with model checking, such that the verification task is improved by e.g. encountering the state-space-explosion problem.

Afterwards, the approach is extended by modular verification techniques inspired by Kupferman and Vardi's module checking [KuVW01]. This allows one to verify specifications for separately compiled Quartz modules [BrSc09] without knowledge about the final environment. This way, (classes of) specifications are identified that are satisfied for a module in an arbitrary context.

In this work, translations from the synchronous MoC to other system representations are defined to allow the use of established tools for their formal verification also for synchronous systems. The input of such transformations will be SGAs, since these systems are conveniently represented by them. In particular, a translation of *SGAs* to *interleaved guarded actions (IGAs)* is presented to make use of tools that are based on the concurrent/asynchronous MoC for systems described by *SGAs*. The idea is to close the gap between *SGAs* and *IGAs* such that tool chains based on these different models can be connected:



To this end, the impact of a system representation on a verification task by embedding a synchronous system differently in a verification tool will be investigated.

The work is evaluated by presenting the AIFProver, a prototype implementation of the presented approach. It integrates interactive verification techniques with model checking.

1.3 Outline

This thesis is structured as follows. Chapter 2 describes the related work for this thesis by giving an overview of the synchronous language Quartz and the corresponding Averest system, introducing some verification techniques and the differences to existing approaches. In Chapter 3 the Hoare calculus is adapted to synchronous programs on source-code level and the encountered problems are discussed. Afterwards, most of these problems are solved by modifying the Quartz representation by defining a normal form. Chapter 4 introduces an approach evolved from the work presented in the previous chapter by defining an interactive verification technique on the compiled SGAs. Afterwards, Chapter 5 extends the approach by introduction of modular verification techniques following the ideas of Kupferman and Vardi's module checking. Afterwards, the impact of the system representation by other MoCs is evaluated in Chapter 6. Then, the feasibility of the presented concept is shown in Chapter 7 by describing the implementation of a interactive verification tool, called AIFProver, which is integrated into the Averest system and approves the thesis by enabling interactive verification for Quartz. Chapter 8 concludes with a summary of the thesis.

Chapter 2

Preliminary and Related Work

This Chapter introduces the synchronous model of computation, the imperative synchronous language Quartz and the Averest system. Moreover, some verification methods and their relation to this work are presented.

Reactive systems were introduced as a special class of systems that have an ongoing interaction with their environment [HaPn85]. Their execution is usually divided into reaction steps, where the system reads inputs from the environment and reacts by computing the corresponding outputs. In contrast to interactive systems, the environment is allowed to initiate the interactions at any time, so that reactive systems usually have to work under real-time constraints. Typical examples are synchronous hardware circuits, protocols, and embedded and cyber-physical systems.

A model of computation is essentially the definition of the execution behavior. It is used to analyze the required computational resources or to discuss the limitations of algorithms. Some examples are Turing machines, finite state machines, lambda calculus, and abstract rewriting systems. In this work, we will concentrate on the classification of sequential, asynchronous and synchronous MoCs, which basically differ in the communication between concurrent executed parts and the execution behavior [Jant04, LeSa98].

2.1 The Synchronous Model of Computation

The sequential MoC describes systems that executes one action/statement after another without any capability of parallel/concurrent execution. The asynchronous MoC [OwGr76a, OwGr76b, ApOl97, RBHH01] represents systems where (independent) actions may or may not be executed in parallel. Hence, the sequential MoC is a subset of the asynchronous MoC.

For the design of reactive systems, synchronous languages have been developed [Halb93, BCEH03, BeGu91, HCRP91, BeCo85] whose paradigm directly reflects the reactive nature of the systems they describe. These languages are based on perfect synchrony [Berr97a] and divide the execution of programs into a discrete sequence of reaction steps that are also called macro steps.

Macro steps consist of finitely many micro steps that are atomic actions of the programs. Within each macro step, the system reads all inputs and instantaneously generates all outputs depending on the current internal states and the read inputs. Also, the next internal state is computed in parallel to the current outputs. Moreover, variables change synchronously between macro steps, i.e. variables have unique values in each macro step.

Macro steps are often viewed as logical time, which is the same in all concurrent threads. For this reason, concurrent threads run in lockstep, which leads to a deterministic form of concurrency. This synchronous concurrency matches with the definition of automata products and parallel compositions of transition systems which is one reason why synchronous languages can be conveniently used for model checking.

2.2 Synchronous Languages

Synchronous programming languages implement the synchronous model of computation. The computation of a synchronous program is partitioned into macro steps that correspond to interactions between the reactive system and its environment. There are imperative languages like Esterel [BeGo92] and Quartz [Schn09] and data-flow oriented languages like Lustre [Halb05], Signal [BBGG85] and Scade [Este04].

2.2.1 Data-flow Oriented Synchronous Languages

Data-flow oriented languages model the system's behavior by equations for output and local variables based on inputs and local variables. Every equation defines the value for a single reaction step. All these languages contain operators for manipulating values in a reaction step and operators for the communication between reaction steps.

Data-flow oriented languages offer implicit parallelism and are obviously well suited for data-processing models.

2.2.2 Imperative Synchronous Languages

Imperative languages like Esterel [BeGo92] and Quartz [Schn09] use a special statement called **`pause`** that separates one macro step from the next one. The control flow may rest at some of these **`pause`** statements and will resume the execution of the micro steps from these locations in the next macro step. The *synchronous model of computation* further demands that all micro steps are executed in zero-time (i.e. within the same variable environment), and all updates to variables are made synchronously for the current macro step. The execution of micro steps within a macro step must therefore be ordered by their data dependencies since it is required that all variables have a unique value per macro step. Thus, the unique assignment to a variable must be done before the variable is read, and thus, there must be no causality/dependency cycles at runtime, which is checked by the compiler during the causality analyses.

The explicit notion of (logical) time also requires different kinds of assignments to enable communication within a step (immediate assignments) and between successive steps (delayed assignments).

In addition to the explicit notion of reaction steps, languages like Esterel and Quartz offer many convenient statements for the design of reactive systems. One class of such statements are pre-emption statements for abortion and suspension that overwrite the normal behavior of the system when a specified condition holds.

2.3 The Synchronous Programming Language Quartz

In the following, we give a brief overview of the synchronous programming language Quartz. This work will concentrate on this synchronous language, because of the rich set of tools available for it. Especially the possibility to compile programs to SGAs is interesting for this work. While SGAs are easy to obtain for Quartz, our work is not only applicable to Quartz. SGAs can be obtained by compilation of many synchronous (and asynchronous) languages following the work of Jens Brandt et al [BGSS12, BrSc11, BrSE12], and thus concentrating on SGAs does not restrict the work to the language Quartz. Hence, the language should be seen as representative of any synchronous language translatable to SGAs. The entire language will not be described here; the interested reader should instead consult [Schn09] for more information.

2.3.1 Core Statements

Figure 2.1 contains a list of core statements of Quartz where it is assumed that S, S_1, and S_2 are statements, ℓ is a location variable, x is a variable, σ is a Boolean expression, α is a type, M is a module's name, *params* is a list of expressions, and all optional parts are enclosed in square brackets.

nothing	(empty statement)
ℓ : **pause**	(start/end of macro step)
$x = \tau$ and **next**(x) $= \tau$	(assignments)
$\{\alpha\ x;\ S\}$	(variable declaration)
if(σ) S_1 **[else** S_2**]**	(conditional)
$S_1; S_2$	(sequential composition)
do S **while**(σ)	(iteration)
$S_1 \parallel S_2$	(synchronous concurrency)
[weak] [immediate] abort S **when**(σ)	(abortion)
[weak] [immediate] suspend S **when**(σ)	(suspension)
M([*params*])	(module call)
assume(φ)	(assumptions)
assert(φ)	(assertions)

Fig. 2.1: List of Quartz Statements

The nothing Statement

The **nothing** statement defines a statement that does neither modify data nor control flow.

The pause Statement

The ℓ: **pause** statement defines a control-flow location ℓ which is a Boolean variable that is true iff the control flow is currently at the statement ℓ: **pause**. Therefore, the **pause** statement separates different macro steps. A **pause** statement is the only statement that consumes time, all other statements are *instantaneous* and are evaluated synchronously.

Control Flow and Data Flow

The control flow can only rest at positions defined by a **pause** statement in the program, since all other statements are executed in zero time, and therefore, the possible control flow states are subsets of the set of locations.

Assignments, Types, and Storage Classes

There are two variants of assignments; and both evaluate the right-hand side in the current macro step. While immediate assignments x=τ immediately transfer the value of τ to the left-hand side x, delayed assignments **next**(x)=τ transfer this value in the following step.

Available types are Booleans, natural numbers (with and without bounds), integers (with and without bounds), bit-vectors (with and without bounds), as well as arrays and tuples of types.

Declarations provide a storage class in addition to the type of a variable. There are two storage classes, namely **mem** and **event** that declare memorized and event variables, respectively. In case no assignment determines the value of a variable, then a memorized variable maintains the value it had in the previous macro step, while an event variable is reset to the default value of the variable's type.

Label variables are event variables, while state and output variables may be event or memorized variables. Both kinds of variables are important for the convenient modeling of reactive systems.

Conditional Statement

The statement **if**(σ) S_1 [**else** S_2] defines a case distinction on the Boolean expression σ and behaves like S_1 in case σ holds and otherwise like the **nothing** statement or the optional statement S_2 if available.

Loop Statements

The **do** S **while**(σ) statement restarts the loop body S in the same macro step when it is completely executed and σ holds, otherwise the whole statement terminates. The **while**(σ) S statement starts the loop body S if σ holds in the first step and restarts it when S is completely executed and σ holds, otherwise the whole statement terminates. An important point for Quartz is that for both loop statements, S must consume time (e.g. at least one **pause** statement must be reached in an execution of the loop's body), otherwise an unwanted infinite loop (inside a single reaction step) would occur.

Sequence Statement

The sequence $S_1;S_2$ defines the subsequent execution of both statements, hence, S_2 is executed right after S_1 is finished. Hence, an instantaneous S_1 implies that S_1 terminates in the macro step it is started and S_2 is started at the same point of time as well. Otherwise, the execution of S_2 is postponed until S_1 terminates.

The Parallel Statement

In addition to the control-flow constructs that are well-known from other imperative languages like conditionals, sequences and loops, Quartz offers synchronous concurrency $S_1 \parallel S_2$. In $S_1 \parallel S_2$, both sub-statements S_1 and S_2 run in lockstep as long as both are active, and the whole parallel statement terminates when the last one of the two sub-statements terminates.

Statements for Pre-emption

Pre-emption statements are another class of statements specific for synchronous languages that allow one to stop or postpone the execution of the contained statement.

The abortion statement **`abort`** S **`when`**(σ) behaves as its body statement S as long as the condition σ is false, and terminates when σ holds. The suspension statement **`suspend`** S **`when`**(σ) also behaves as its body statement S as long as the condition σ is false, and postpones the computation in each step where σ holds. Both pre-emption statements can moreover be weak or strong which makes a difference on their influence on the control and data flow of the controlled statement S: The weak variants allow all actions of the data flow to take place even at the time of pre-emption, while the strong variant forbids them (at the time of pre-emption). Nevertheless, both variants influence the control flow (depending on the kind of pre-emption statement) in the same way. Additionally, there exist an immediate and delayed variant of each pre-emption statement. The immediate forms do already check for pre-emption at starting time of the preemption statement, while the default is to check the pre-emption in the macro steps after the first one.

In Figure 2.2 the four possible abort behaviors are presented, the solid arrows on the left-hand side represent the (strong) abort behavior adding the dashed line to the behavior leads to the (strong) immediate abort behavior. The arrows on the right-hand side represents the weak variants. As can be seen, the strong aborts take place at the *beginning* of a macro step, while the weak aborts take place at the *end* of a macro step.

In Figure 2.3, the four possible suspend behaviors are depicted. The left-hand side represents again the strong versions and the right-hand side the weak versions. The dashed lines are added for the immediate versions of suspend as well.

Module Declarations

A Quartz module defines an interface with inputs (indicated by ?), outputs (indicated by !) and inouts (without ! or ?) and a behavior for this interface by a statement for the module body. The body statement of the module is only allowed to read its input variables and to write to its output variables. Each module must be declared in a separate Quartz file, but may call other modules. There must be an hierarchical order of all modules, because cyclic and recursive calls are not allowed.

The Module Call

A module call consists of the called module's name and a list of parameter expressions that have to match the number and type of the module's interface. During linking, the module

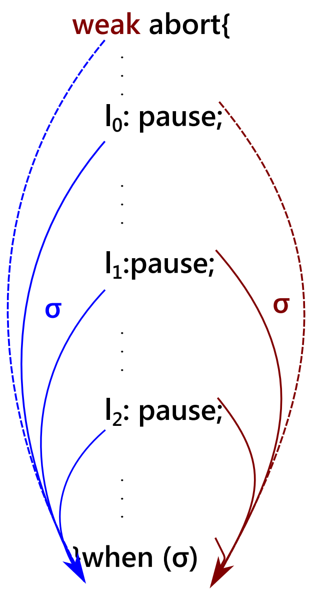


Fig. 2.2: Abort Behavior

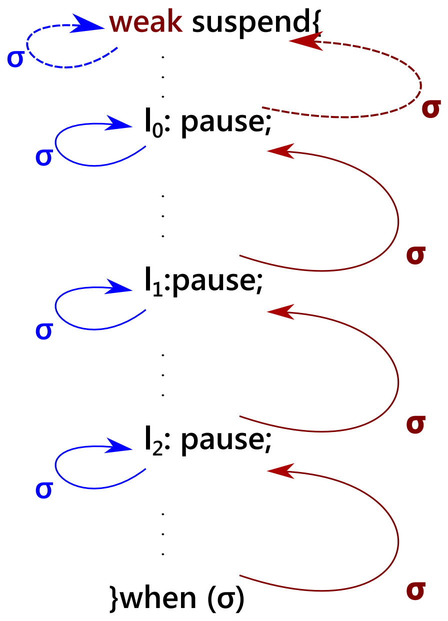


Fig. 2.3: Suspend Behavior

call is replaced by the module's body and the input parameters are replaced with expressions of the same type, output parameters are replaced with local or output variables of the calling module, and thus, the assignments of the called modules then become assignments of the calling module. Of course, the calling module may also make assignments to its local and output variables, so that the two behaviors are combined.

Assumptions and Assertions

Assumptions and assertions are statements mainly used for verification purposes. The Quartz compiler adds automatically assertions to detect certain errors, like overflow or underflow for bounded number representations, array out-of-bound accesses and division by zero. Additionally, the user is allowed to add assumptions and assertions in the code as well. Assertions have to be verified and assumptions are precondition that are assumed by the compiler.

2.3.2 Additional Macro Statements

Besides the core statements, there are more statements that can be reduced to core statements. The simplest one is the loop statement that represents an infinite loop:

loop{S} ≡ **while** (**true**) {S}

This statement is used to emphasize that the loop does not terminate.

Additionally, reactive systems need to wait for certain conditions. The simplest way to do that is with the **await** statement. Similar to the preemption statements, there exists an immediate and a delayed variant. The behavior is defined as given in Figure 2.4 and Figure 2.5.

```
await(c):=do{ pause;} while(¬c);
```

Fig. 2.4: Await Behavior

```
immediate await(c):=while(¬c){ pause;};
```

Fig. 2.5: Immediate Await Behavior

The **await** statement contains a **pause** statement, hence it is possible to annotate the statement to rename the contained **pause**. Additionally, the **await** represents the end or the beginning of a reaction step.

Quartz-Code Example

```
module ABRO(event ?a,?b,?r,!o) {
   loop
      abort {
         wa: await(a);
         ||
         wb: await(b);
         emit(o);
         s0: assert(a ∨ b);
         wr: await(r);
      } when(r);
} satisfies {
   s1 : assert A G (o → a ∨ b);
   s2 : assert A G (o → X ¬o);
   s3 : assert A G (o→([a B⃖ r]∧[b B⃖ r]));
}
```

Fig. 2.6: Quartz Module ABRO

	1	2	3	4	5	6
a	F	T	F	T	F	T
b	F	F	T	T	T	F
r	F	F	F	F	T	F
wa	F	T	F	F	F	T
wb	F	T	T	F	F	T
wr	F	F	F	T	T	F
o	F	F	T	F	F	F

Fig. 2.7: Example Behavior

Figure 2.6 shows the source code of a simple Quartz module called ABRO with a contained assertion and three specifications given in temporal logic and an example behavior in Figure 2.7. The module has three Boolean inputs `a,b,r` (indicated by `?`) and a Boolean output `o` (indicated by `!`). The module waits on the input signals `a` and `b` in parallel, and immediately emits output `o` as soon as the last one of `a` and `b` occurred. This behavior is restarted if `r` occurs.

Assertion `s0` demands that either `a` or `b` must hold in the macro step, when `o` is emitted, and specification `s1` essentially states the same from a global point of view. Specification `s2` asserts that `o` cannot hold at two successive points of time. The last specification uses past-temporal operators stating that emitting `o` requires that `a` and `b` occurred in between the last time where `r` held and the current step where `o` holds.

The example behavior in Figure 2.7 shows that `o` is emitted in the third macro step, because the event `b` holds in the same macro step and `a` already held in the second macro step. After emitting `o`, the system waits for the reset signal indicated by `wr` in the fourth and fifth macro step. Additionally, the event `r` holds in the fifth macro step, which leads to a restart of the system's behavior in the following macro step.

2.3.3 Schizophrenia and Causality

As mentioned above, synchronous programs offer many advantages for system design, but they also challenge the compilers [SBST05a]: In particular, schizophrenia and causality problems must be addressed [Schn09, Berr99, ScBS06].

Schizophrenia

Schizophrenia issues may occur if a statement is (re-)started at the time of its termination so that the compiler has to keep track of different incarnations of the statement. In particular, local variable declarations have to be handled with some care to distinguish values of variables that belong to the same variable but to different scopes.

The problem is that following the synchronous MoC all variables must have a unique value during a macro step. Executing an additional local declaration for a variable `x` in a macro step where variable `x` is still valid requires a new instance of this variable and the read and write accesses to both variables have to be distinguished accordingly. The target variable of an assignment is uniquely determined by the scope of the variable. In particular, schizophrenia problems are easily solvable by copying variables or parts of the program [Schn09, Mali94, Berr97a] and handled during compilation by the Averest compiler.

A simple example containing a schizophrenic statement is given in Figure 2.8. There, the local declaration for variable `c` (Line 4) is executed by re-entering the loop body while the variable `c` from the previous execution of the loop body is still valid and also used to assign the variable `a` a value. The introduction of a new instance of `c` prevents an (incorrect) data-dependent cycle for `c` and `a`. The variable `c` in the equations `c=a` (Line 5) and `a=c+1` (Line 8) are different, because they are encountered in different iterations of the loop.

Causality

Causality problems occur if a statement modifies variables that are responsible for triggering the execution or evaluation of the statement. Hence, a statement may disable its execution or it may justify its execution, which are both unwanted behaviors. The causality analysis, which is a fixpoint iteration over a variable environment, ensures that a program is causally correct. The iteration starts with the environment where all variables are unknown except for the inputs and the variables written by delayed assignments in the previous step. Then, the variables are evaluated with respect to the current environment and update the partial environment until no further progress is seen. Thus, the number of iteration steps is bounded by the number of variables. A detailed description of causality and the corresponding causality analysis is given in [Schn09, ScBS04b, BrSc08a, Berr00, Berr99].

```
module Schizo(nat{4} a) {
   a = 1;
   loop {
      nat{4} c;
      c = a;
      next(c) = c+2;
      w: pause;
      a = c+1;
   }
} satisfies {
 s1 : assert (a==1);
 s2 : assert A X (a==4);
 s3 : assert A X X (a==7);
}
```

Fig. 2.8: Example of Schizophrenia

The causality analysis has to determine that the statements of a synchronous program can be executed in a causal order where all trigger conditions are determined before the corresponding statements are executed and a written value/variable is not overwritten. Hence, a program is causally correct iff for all variables unique values are derived (by the causality analysis sketched above) for all reachable states and all possible inputs, such that for all possible inputs, the outputs can be determined in an (dynamic) schedule that respects the data dependencies.

The Figures 2.9 and 2.10 contain Quartz programs with causality conflicts. The first program does not have any behavior, because both possible values for the Boolean variable `a` lead to a contradiction: in case `a=true` the **`if`** statement is not entered, hence no assignment sets `a`, in the other case (`a=false`) the **`if`** statement is entered, hence the contained assignment sets `a`. The other example has too many behaviors: in case `a=true` the assignment contained in the **`if`** statement sets `a` and leads to a valid execution, in the other case (`a=false`) no assignment is executed, which leads again to a valid execution. Hence the behavior is therefore an unwanted non-deterministic behavior.

```
module causality1(event a) {
   if (!a) emit(a);
}
```

Fig. 2.9: Causality Problem #1

```
module causality2() {
   event a;
   if (a) a=true;
}
```

Fig. 2.10: Causality Problem #2

2.3.4 Formal Semantics

The semantics of Quartz is defined as operational semantics following Plotkin's structural approach [Moss06, Plot81]. In [Schn09] the *structural operational semantics (SOS)* rules for all Quartz statements are defined. Each rule defines the behavior of a statement under given assumptions, an environment $\mathcal{E}$ that defines value for each variable, and the incarnation index function $\hbar$ (see [Schn09] for more details) that maps each variable to its incarnation index, i. e. the number of scopes of the variable that have been entered in this macro step so far. The incarnation index function is used to solve schizophrenia issues.

Given a complete environment $(\mathcal{E},\hbar)$ of the current macro step, a statement S and a precondition α, these SOS rules are of the form:

$$\frac{\alpha}{\langle \mathcal{E},\hbar,S\rangle \twoheadrightarrow \left\langle \hbar',S',\mathcal{D},t\right\rangle}$$

The semantics of these rules is that given that α holds, the rule computes the updated incarnation index function $\hbar'$, a residual statement S' that has to be executed in the next macro step, a set of pairs $\mathcal{D}$ representing the delayed assignments executed in the current step by S and finally the Boolean value t indicating the time consumption of S ($t = \mathsf{true}$ means S is instantaneous).

Examples

In the following one of the rules for the if-then-else statement **`if`** `(`σ`)` $\mathtt{S}_1$ **`else`** $\mathtt{S}_2$ is given:

$$\frac{[\![\sigma]\!]_{\mathcal{E}}^{\hbar} = \mathsf{true} \;\text{ and }\; \langle \mathcal{E},\hbar,S_1\rangle \twoheadrightarrow \left\langle \hbar_1',S_1',\mathcal{D}_1,t_1\right\rangle}{\langle \mathcal{E},\hbar,\mathbf{if}(\sigma)S_1\;\mathbf{else}\;S_2\rangle \twoheadrightarrow \left\langle \hbar_1',S_1',\mathcal{D}_1,t_1\right\rangle}$$

This rules states that the behavior is defined by the then-part of the statement iff the condition σ holds in the environment $\mathcal{E}$. The SOS rule for the parallel statement $\mathtt{S}_1 \parallel \mathtt{S}_2$:

$$\frac{\langle \mathcal{E},\hbar,S_1\rangle \twoheadrightarrow \left\langle \hbar_1',S_1',\mathcal{D}_1,t_1\right\rangle \qquad \langle \mathcal{E},\hbar,S_2\rangle \twoheadrightarrow \left\langle \hbar_2',S_2',\mathcal{D}_2,t_2\right\rangle}{\langle \mathcal{E},\hbar,S_1 \parallel S_2\rangle \twoheadrightarrow \left\langle \mathsf{Max}(\hbar_1',\hbar_2'),S_1' \parallel S_2',\mathcal{D}_1 \cup \mathcal{D}_2,t_1 \wedge t_2\right\rangle}$$

This rules states that the behavior is defined by the given combination of the two evaluated statements. The set of delayed assignments and the t's are joined. Furthermore, the remaining statements S_1' and S_2' are again executed in parallel in the next step.

2.3.5 Compilation to Synchronous Guarded Actions

Synchronous languages can be compiled to Synchronous Guarded Actions (*SGAs*) that are a convenient intermediate representation for compilers and can represent each Quartz program by preserving its original semantics [Schn09, Schn01a, Schn02]. Hence, the SOS semantics is followed. In [ScBS06] the correctness of this compilation technique was proven. *SGAs* have a very simple structure:

Definition 1 (Synchronous Guarded Actions). *Given a set of variables $\mathcal{V}$, a* SGA *is a pair $\gamma \Rightarrow \alpha$ consisting of a Boolean condition γ called the trigger and its action α. An action is either an immediate assignment* `x` $= \tau$*, a delayed assignment* **`next`**`(x)` $= \tau$*, an assumption* **`assume`**(φ)*, or an assertion* **`assert`**(φ)*, where φ is a Boolean formula over $\mathcal{V}$, τ is an expression of* `x`*'s type, and* `x` $\in \mathcal{V}$.

Any synchronous system can be represented by a set of synchronous guarded actions. The synchronous model of computation demands that in each macro step, *all enabled guarded actions have to be executed and that their execution does not invalidate once enabled guards (so that causality will be respected).* Clearly, to deal with causality problems, there must be a partial order to execute the guarded actions so that the variables become known before (in terms of micro steps) they are read. These considerations are major problems for compilers and code generators and will be discussed in detail in Chapter 6.

Intuitively, the behavior of *SGAs* is that whenever the guard is true in a macro step s, the action is fired, which means that the corresponding equation must be true. In case of an immediate assignment `x` $= \tau$ this means that on s (in the current step), variable `x` must have the same value as τ, and for a delayed assignment **`next`**`(x)` $= \tau$, it means that in the following step s', variable `x` must have the value that τ has on s. Assumption and assertions are used for verification purposes only and become assumptions and proof obligations, respectively.

In addition to the generated guarded actions, the behavior of a system moreover includes for every variable an implicit guarded action which is called the default reaction: It defines the value of a variable in case that no action has determined its value in the current macro step (obviously, this is the case iff the guards of all immediate assignments in the current step, and the guards of delayed assignments in the previous step of a variable are false). The default reaction depends on the variables storage type: Memorized variables keep their value from the previous step while event variables are reset to the default value of the variable's type.

Furthermore, we partition the set of guarded actions into *control- and data-flow* actions, which will be important for defining strong and weak pre-emption.

Definition 2 (Control and Data Flow). *The control flow consists of guarded actions writing a label variable $l \in \mathcal{V}_l$, while the data flow are guarded actions writing a local variable $\mathcal{V}_s$ or an output $\mathcal{V}_o$. Following the Averest's compiler output, we assume that guarded actions of the control flow have the form $\gamma \Rightarrow$* **`next`**$(\ell) = \mathsf{true}$.

Label variables $\mathcal{V}_l$ correspond with places in the program where the control flow can rest between the macro steps, i.e., these labels denote places in the program code where a macro step ends and where another one starts (i.e. **`pause`** statements). Labels are translated to Boolean variables where only guarded actions as shown above are obtained. Since labels are event variables, they will be automatically reset to false if there is no assignment setting them to true.

Since the **`pause`** statements (which are the essential control flow locations) have names that occur in the guarded actions as Boolean variables, there is a close relationship between the guarded actions and the original source code.

An example for generated *SGAs* of a Quartz program is presented in the following section. In [BrSc09, Schn09] the general compilation approach is described in full detail.

Example

```
system ABRO :
interface :
    a, b, r : input event bool
          o : output event bool
locals :
    init, wa, wb, wr : label bool
synchronous guarded actions :
    control flow:
        init  ⇒ next(wa) = True
        init  ⇒ next(wb) = True
        ¬r∧wa∧¬a∨r∧(wr∨wa∨wb) ⇒ next(wa) = True
        ¬r∧wb∧¬b∨r∧(wr∨wa∨wb) ⇒ next(wb) = True
        ¬r∧(wr∨a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa)
          ⇒ next(wr) = True
    data flow:
        ¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa) ⇒ o = True
    assertions:
        ¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa) ⇒ assert(a ∨ b)
specifications:
    s1: A G o → a ∨ b
    s2: A G o → X ¬o
```

Fig. 2.11: AIF System for ABRO

The ABRO module (given in Figure 2.6) is compiled to the guarded actions shown in Figure 2.11, which are separated into control flow, data flow, and assertions. Note that a new control flow location **init** has been automatically added which is often called the *boot location*. This allows one to distinguish the starting point (only label **init** holds) from the termination point (no label holds). The relation of the source code and the labels in the *SGAs* is also seen in this example, e.g. `wr` is reached by not reading `r` and either leaving `wa` and `wb` (by reading `a` and `b`) as well as the simple cases where only `wa` (`wb`) holds and `a` (`b`) is read or by already being at `wr`.

The example behavior in Figure 2.7 is computable by applying the SOS rules on the source code or evaluating the equivalent set of *SGAs* (derived by the compiler from the source code by following the semantics/SOS rules).

2.3.6 Translation to Transition Systems

In this section, we describe how symbolic representations of transition systems (Kripke structures) [Schn03] can be directly generated from a set of guarded actions (that has been obtained by the compilation of a synchronous module). The definitions are restricted to the Boolean type $\mathbb{B}$ for the sake of simplicity.

Definition 3 (Transition Systems). *A* transition system $\mathcal{T} = (\mathcal{S},\mathcal{I},\mathcal{R},\mathcal{L})$ *for a finite set of variables* $\mathcal{V}$ *is given by a finite set of states* $\mathcal{S}$*, a set of initial states* $\mathcal{I} \subseteq \mathcal{S}$*, a transition relation* $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$*, and a label function* $\mathcal{L} : \mathcal{S} \to \mathbb{B}^{\mathcal{V}}$ *that maps each state to a variable assignment* $\mathcal{L}(s) : v \to \mathbb{B}$*.*

Our aim is to generate Boolean formulas $\varphi_{\mathcal{I}}$ and $\varphi_{\mathcal{R}}$ for the initial state condition and the transition relation of a transition system that can be directly used for model checking. To this end, we have to use for each variable of the program a corresponding variable that denotes the value in the next state of the program:

Definition 4 (Next State Variables). *The value of a variable* $\mathtt{x} \in \mathcal{V}$ *in the next macro step/state is denoted by* $\mathtt{x}' \in \mathcal{V}'$*. Hence,* **next**(x)=$\tau$ *assigns the variable* x*' the value* τ*.*

The expression $[\![\varphi]\!]_{\vartheta}$ is used in the following and denotes the evaluation[1] of a propositional formula φ under a truth assignment represented by $\vartheta \subseteq \mathcal{V}$, i.e.,

- $[\![x]\!]_{\vartheta} :\Leftrightarrow x \in \vartheta$ for every variable $x \in \mathcal{V}$
- $[\![\neg\varphi]\!]_{\vartheta} :\Leftrightarrow \neg [\![\varphi]\!]_{\vartheta}$
- $[\![\varphi \wedge \psi]\!]_{\vartheta} :\Leftrightarrow [\![\varphi]\!]_{\vartheta} \wedge [\![\psi]\!]_{\vartheta}$
- $[\![\varphi \vee \psi]\!]_{\vartheta} :\Leftrightarrow [\![\varphi]\!]_{\vartheta} \vee [\![\psi]\!]_{\vartheta}$

The formulas $\varphi_{\mathcal{I}}$ and $\varphi_{\mathcal{R}}$ over some set of variables[2] $\mathcal{V}$ then encode the following state transition system $\mathcal{T} = (\mathcal{S},\mathcal{I},\mathcal{R},\mathcal{L})$:

- states $\mathcal{S}$
- initial states $\mathcal{I} := \{s \in \mathcal{S} \mid [\![\varphi_{\mathcal{I}}]\!]_s = \mathsf{true}\}$
- transitions $\mathcal{R} := \{(s,s') \in \mathcal{S} \times \mathcal{S}' \mid [\![\varphi_{\mathcal{R}}]\!]_{s,s'} = \mathsf{true}\}$
- label function $\mathcal{L}(s) = s$, i.e., the state $s \in \mathcal{S}$ is the variable assignment $\mathcal{L}(s)$.

As seen above, we identify states with their label function. After these general explanations of the notation used below, we now consider how to define formulas $\varphi_{\mathcal{I}}$ and $\varphi_{\mathcal{R}}$ for a given set of guarded actions. To this end, we only consider guarded actions whose actions are assignments, since only these guarded actions determine the behavior of a module. The remaining actions, i.e., assumptions and assertions, are used to construct specifications, and do therefore not modify the underlying transition system.

The above informal remarks lead to the following formal definition of a state transition system. The aim is to generate Boolean formulas for the initial state condition and the transition relation that can be directly used for model checking. We start by defining some auxiliary functions.

Definition 5 (Reactions per Variable). *Assume that for a Boolean*[3] *variable* $\mathtt{x} \in \mathcal{V}$*, we have the guarded actions with immediate assignments* $(\gamma_1,\mathtt{x} = \tau_1),\ldots,(\gamma_p,\mathtt{x} = \tau_p)$ *and with delayed assignments* $(\chi_1,$**next**$(\mathtt{x}) = \pi_1),\ldots,(\chi_q,$**next**$(\mathtt{x}) = \pi_q)$*. Then, we define the following Boolean formulas over* $\mathcal{V} \cup \mathcal{V}'$*, where* $\mathsf{Initial}(\mathtt{x})$ *denotes the initial value of variable*

[1] The reader may excuse the sloppy use of propositional logic operators here that are used both at the meta-level and the level of propositional formulas.

[2] This means that $\varphi_{\mathcal{R}}$ has occurrences of variables $x \in \mathcal{V}$ and corresponding variables x' while $\varphi_{\mathcal{I}}$ has only occurrences of variables $x \in \mathcal{V}$.

[3] Other types must be represented by several Boolean variables

x *that depends on x's type and is e.g.* false *for Booleans. Additionally, we make use of the substitution* $\langle\varphi\rangle_{\mathcal{V}}^{\mathcal{V}'}$ *that replaces all occurrences of a variable* $\mathtt{v} \in \mathcal{V}$ *in* φ *by the corresponding variable* $\mathtt{v}' \in \mathcal{V}'$*.*

- $\mathsf{Default}(\mathtt{x}) := \begin{cases} \mathsf{Initial}(\mathtt{x}) & \textit{: if } \mathtt{x} \textit{ is an event variable} \\ \mathtt{x} & \textit{: if } \mathtt{x} \textit{ is a memorized variable} \end{cases}$
- $\mathsf{ImmActs}(\mathtt{x}) := \bigwedge_{j=1}^{p}(\gamma_j \to \mathtt{x} = \tau_j)$
- $\mathsf{DelActs}(\mathtt{x}) := \bigwedge_{j=1}^{q}(\chi_j \to \mathtt{x}' = \pi_j)$
- $\mathsf{InitDefActs}(\mathtt{x}) := \left(\bigwedge_{j=1}^{p} \neg\gamma_j\right) \to \mathtt{x} = \mathsf{Initial}(\mathtt{x})$
- $\mathsf{NextDefActs}(\mathtt{x}) := \left\langle \bigwedge_{j=1}^{p} \neg\gamma_j \right\rangle_{\mathcal{V}}^{\mathcal{V}'} \wedge \left(\bigwedge_{j=1}^{q} \neg\chi_j\right) \to \mathtt{x}' = \mathsf{Default}(\mathtt{x})$

Using the above formulas, an initial state condition $\mathcal{I}$ and the transition relation $\mathcal{R}$ of a transition system can be constructed as follows:

Definition 6 (Symbolic Representation of Systems). *For a synchronous system over the variables* $\mathcal{V}$ *consisting of inputs* $\mathcal{V}_i$*, labels and local variables* $\mathcal{V}_l$*, and output variables* $\mathcal{V}_o$*, the transition system* $\mathcal{T} := (\mathcal{S},\mathcal{I},\mathcal{R},\mathcal{L})$ *is defined by the states* $\mathcal{S} = 2^{\mathcal{V}}$*,* $\mathcal{L}(s) := s$*, the following initial state condition* $\mathcal{I}$*, and the state transition relation* $\mathcal{R}$*.*

- $\mathcal{I} := \bigwedge_{\mathtt{x}\in(\mathcal{V}_l\cup\mathcal{V}_o)} \mathsf{ImmActs}(\mathtt{x}) \wedge \bigwedge_{\mathtt{x}\in(\mathcal{V}_l\cup\mathcal{V}_o)} \mathsf{InitDefActs}(\mathtt{x})$
- $\mathcal{R} := \bigwedge_{\mathtt{x}\in(\mathcal{V}_l\cup\mathcal{V}_o)} \mathsf{ImmActs}(\mathtt{x}) \wedge \bigwedge_{\mathtt{x}\in(\mathcal{V}_l\cup\mathcal{V}_o)} \mathsf{DelActs}(\mathtt{x}) \wedge \bigwedge_{\mathtt{x}\in(\mathcal{V}_l\cup\mathcal{V}_o)} \mathsf{NextDefActs}(\mathtt{x})$

Whenever one of the guards γ_i of an immediate assignment $\gamma_i \Rightarrow \mathtt{x} = \tau_i$ holds in the definition of $\mathcal{R}$, then the equation $\mathtt{x} = \tau_i$ must hold, since the assignment has an immediate effect. Analogously, if a guard χ_i of a delayed assignment $\chi_i \Rightarrow \mathbf{next}(\mathtt{x}) = \pi_i$ holds, then the equation $\mathtt{x}' = \pi_i$ that defines the value for x in the next step by the current step's value of π_i must hold. The value of x is determined by the default action ($\mathsf{Default}(\mathtt{x})$) if no guard χ_i held in the previous step and no guard γ_i holds in the current step.

Example

Reconsidering the example module `ABRO`, the following formulas are obtained where the formulas are split into the computed guarded actions and the default actions. The initial condition $\mathcal{I}_{\mathsf{cl}}$ contains the guarded action that sets `o` to `true`, the implicit assignment to the boot flag and to the label variables, and the default reaction in case the assignment to `o` does not take place. This condition is used to define the initial states. The transition relation $\mathcal{R}_{\mathsf{cl}}$ defines the transition from one state to another. It contains all guarded actions of the variables and labels. Additionally, it contains the implicit assignment to the boot flag and the default reaction for the labels and variables.

$$\begin{array}{rl}
\mathcal{I}_{\mathsf{cl}} := & (\neg\mathtt{r}\wedge(\mathtt{a}\wedge\mathtt{wa}\wedge\mathtt{b}\wedge\mathtt{wb}\vee\neg\mathtt{wa}\wedge\mathtt{b}\wedge\mathtt{wb}\vee\neg\mathtt{wb}\wedge\mathtt{a}\wedge\mathtt{wa})\rightarrow\mathtt{o}=\mathsf{true}) \\
\wedge & \left(\begin{array}{l}
\mathsf{true}\rightarrow\mathbf{init}=\mathsf{true}\wedge \\
\mathsf{true}\rightarrow\mathtt{wa}=\mathsf{false}\wedge \\
\mathsf{true}\rightarrow\mathtt{wb}=\mathsf{false}\wedge \\
\mathsf{true}\rightarrow\mathtt{wr}=\mathsf{false}\wedge \\
\mathtt{r}\vee(\neg\mathtt{a}\vee\neg\mathtt{wa}\vee\neg\mathtt{b}\vee\neg\mathtt{wb})\wedge(\mathtt{wa}\vee\neg\mathtt{b}\vee\neg\mathtt{wb})\wedge(\mathtt{wb}\vee\neg\mathtt{a}\vee\neg\mathtt{wa})\rightarrow\mathtt{o}=\mathsf{false}
\end{array}\right) \\
\mathcal{R}_{\mathsf{cl}} := & \left(\begin{array}{l}
\mathbf{init}\vee\neg\mathtt{r}\wedge\mathtt{wa}\wedge\neg\mathtt{a}\vee\mathtt{r}\wedge(\mathtt{wr}\vee\mathtt{wa}\vee\mathtt{wb})\rightarrow\mathbf{next}(\mathtt{wa})=\mathsf{true}\wedge \\
\mathbf{init}\vee\neg\mathtt{r}\wedge\mathtt{wb}\wedge\neg\mathtt{b}\vee\mathtt{r}\wedge(\mathtt{wr}\vee\mathtt{wa}\vee\mathtt{wb})\rightarrow\mathbf{next}(\mathtt{wb})=\mathsf{true}\wedge \\
\neg\mathtt{r}\wedge\mathtt{wr}\vee\neg\mathtt{r}\wedge(\mathtt{a}\wedge\mathtt{wa}\wedge\mathtt{b}\wedge\mathtt{wb}\vee\neg\mathtt{wa}\wedge\mathtt{b}\wedge\mathtt{wb}\vee\neg\mathtt{wb}\wedge\mathtt{a}\wedge\mathtt{wa})\rightarrow\mathbf{next}(\mathtt{wr})=\mathsf{true}\wedge \\
\neg\mathtt{r}\wedge(\mathtt{a}\wedge\mathtt{wa}\wedge\mathtt{b}\wedge\mathtt{wb}\vee\neg\mathtt{wa}\wedge\mathtt{b}\wedge\mathtt{wb}\vee\neg\mathtt{wb}\wedge\mathtt{a}\wedge\mathtt{wa})\rightarrow\mathtt{o}=\mathsf{true}
\end{array}\right) \\
\wedge & \left(\begin{array}{l}
\mathsf{true}\rightarrow\mathbf{next}(\mathbf{init})=\mathsf{false}\wedge \\
\neg\mathbf{init}\wedge(\mathtt{r}\vee\neg\mathtt{wa}\vee\mathtt{a})\wedge(\neg\mathtt{r}\vee\neg\mathtt{wr}\wedge\neg\mathtt{wa}\wedge\neg\mathtt{wb})\rightarrow\mathbf{next}(\mathtt{wa})=\mathsf{false}\wedge \\
\neg\mathbf{init}\wedge(\mathtt{r}\vee\neg\mathtt{wb}\vee\mathtt{b})\wedge(\neg\mathtt{r}\vee\neg\mathtt{wr}\wedge\neg\mathtt{wa}\wedge\neg\mathtt{wb})\rightarrow\mathbf{next}(\mathtt{wb})=\mathsf{false}\wedge \\
\mathtt{r}\vee((\neg\mathtt{a}\vee\neg\mathtt{wa}\vee\neg\mathtt{b}\vee\neg\mathtt{wb})\wedge(\mathtt{wa}\vee\neg\mathtt{b}\vee\neg\mathtt{wb})\wedge(\mathtt{wb}\vee\neg\mathtt{a}\vee\neg\mathtt{wa}))\rightarrow\mathbf{next}(\mathtt{wr})=\mathsf{false}\wedge \\
\mathtt{r}\vee((\neg\mathtt{a}\vee\neg\mathtt{wa}\vee\neg\mathtt{b}\vee\neg\mathtt{wb})\wedge(\mathtt{wa}\vee\neg\mathtt{b}\vee\neg\mathtt{wb})\wedge(\mathtt{wb}\vee\neg\mathtt{a}\vee\neg\mathtt{wa}))\rightarrow\mathtt{o}=\mathsf{false}
\end{array}\right)
\end{array}$$

2.3.7 Alternative Representations of Synchronous Systems

There are many representation techniques for synchronous programs [PoEB07] like generation of equations or an (explicit) transition diagram. These two kinds are somehow extreme cases, where the size of the equations is polynomial and generates a symbolic description of the program's behavior, while the size of the transition diagram is exponential and generates an explicit enumeration of the possible program states and reactions. In the following, it is shown how to generate equations. Afterwards, *extended finite state machines* (EFSMs) are presented that are a compromise between the extreme cases for representation: An EFSM explicitly enumerates the control flow states (similar to the transition diagram representation) while describing the data flow in a symbolic manner (similar to the equations). In general, its size is therefore also exponential, but in practice, this problem is rarely observed.

Equations

As already mentioned, it is possible to translate synchronous guarded actions into an equation system by aggregating all guarded actions on the same variable (and the corresponding default reaction). This is useful for theorem proving since one can then use the equation system as rewrite system to prove the specification. It is usually not trivial to generate such an equation system, but a corresponding transformation is already implemented in the Averest system. The translation of *SGAs* to equations will generate exactly one equation for each output, local and label variable. Furthermore, an additional carrier variable must be added for each variable to which an immediate and delayed assignment is made. The carriers will simply hold the value until the next point of time and have the same behavior as the variable $\mathtt{x}' \in \mathcal{V}'$ for $\mathtt{x} \in \mathcal{V}$ in the transition diagram representation. All carrier variables and all variables written only by delayed assignments require an additional **`init`** value for the first macro step (see [Schn09] for details).

The execution of an equation system under the synchronous model of computation is then as follows: in every macro step new input variables are read and all of the equations are evaluated with regards to the newly read values. The resulting right-hand side of each equation is then assigned to its respective left-hand-side variable.

The equation system for the ABRO example generated by the aif2aif tool of the Averest system is shown in Figure 2.12. All guarded actions writing the same variables and the corresponding default reaction are composed together. Hence, every variable is defined by a single equation that must be evaluated in each step. Additionally, the initial values for the delayed written labels, which is false is implicitly given.

```
control flow:
    True ⇒ next(init) = False
    True ⇒ next(wb) = ¬r∧wb∧¬b ∨ r∧(wr∨wa∨wb) ∨ init
    True ⇒ next(wa) = ¬r∧wa∧¬a ∨ r∧(wr∨wa∨wb) ∨ init
    True ⇒ next(wr) = ¬r∧(wr ∨ a∧wa∧b∧wb ∨ ¬wa∧b∧wb ∨ ¬wb∧a∧wa)
data flow:
    True ⇒ o = ¬r∧(a∧wa∧b∧wb ∨ ¬wa∧b∧wb ∨ ¬wb∧a∧wa)
```

Fig. 2.12: Equations for ABRO

Extended Finite State Machines (EFSMs)

Another important transformation is the computation of an extended finite state machine (EFSM). EFSMs are frequently used as a convenient form to describe programs with potentially infinite data types. Additionally, this description could be useful for interactive verification in that one can decompose the proof goal with respect to the reachable control flow states.

A formal definition of an EFSM is as follows:

Definition 7. *An extended finite state machine (EFSM) is a tuple $\mathcal{A} = (\mathcal{Q},\mathcal{I},\mathcal{R},\mathcal{P})$ where $\mathcal{Q}$ is a finite set of states, $\mathcal{I} \subseteq \mathcal{Q}$ is the set of initial states, and $\mathcal{R} \subseteq \mathcal{Q} \times \mathcal{P} \times \mathcal{Q}$ is the transition relation, where $\mathcal{P}$ is the set of pairs (Φ, S) where Φ is a Boolean condition and S is a set of guarded actions.*

The meaning of a transition labeled with (Φ, S) is as follows: this transition is enabled iff the condition Φ holds. In that case, the transition is taken (note that there is no non-determinism, the synchronous model of computation ensures that at every state only one transition is enabled), and the actions in S are immediately and synchronously executed in that all actions are executed whose guard holds.

Example

The EFSM of the ABRO module is shown in Figure 2.13. Obviously, we can associate every state of the EFSM by the set of active control flow locations in that state. For example, the states 1, 3, and 4 represent the three different possibilities for the control flow being in the parallel statement, i.e., either the control flow is in both threads (waiting for `a` and `b`), or one thread already terminated (by reading `a` or `b`) and is waiting for the other one to terminate.

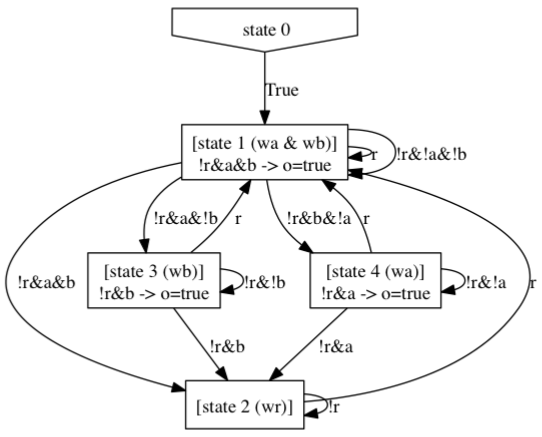


Fig. 2.13: EFSM for ABRO

2.4 Averest

As a long term project, the embedded systems group of the university of Kaiserslautern implements their research results [BaBS10, BaBS11b, Gemu13a, Baue12, Bran13, GeBS14] in the Averest system (see `http://www.averest.org`) for modeling, simulation, synthesis, verification and HW/SW-codesign. It uses the Esterel-like synchronous language Quartz to model synchronous/reactive systems [BCEH03, BeGo92].

Design Flow

The design flow used in Averest is shown in Figure 2.14. It contains a compiler that computes for a given synchronous Quartz program an equivalent set of *SGAs* that are stored in *Averest Interchange Format (AIF)* files. Using an embedding of the Quartz language in the theorem prover HOL [Schn01a, Schn02], Schneider et al. verified the correctness of the compile algorithm, i.e. that the semantics of the program is preserved during the translation [Schn01a, Schn02, ScBS06, Schn09]. Since it is possible to separately compile Quartz modules [BrSc09, Schn09, ScBS06] and to link them afterwards to AIF systems, one has to distinguish between modules and linked systems. Modules can be called in other Quartz modules so that one is able to create libraries for later reuse.
There are several transformations available to modify a generated AIF system description. For example, the partitioning of compound data types like tuples and arrays to scalar types, reduction to Boolean types for hardware synthesis, the aggregation of all guarded actions on one variable into a single guarded action (so that equations are generated), dead code elimination, the generation of an EFSM, and many more are available.

After suitable transformations, AIF systems can be converted to various target languages. For example, there are code generators for software synthesis (producing C, Java or SystemC) or hardware synthesis (producing VHDL and Verilog files). Moreover, a simulator and a code generator for the well-known model checker SMV are also available in the Averest framework.

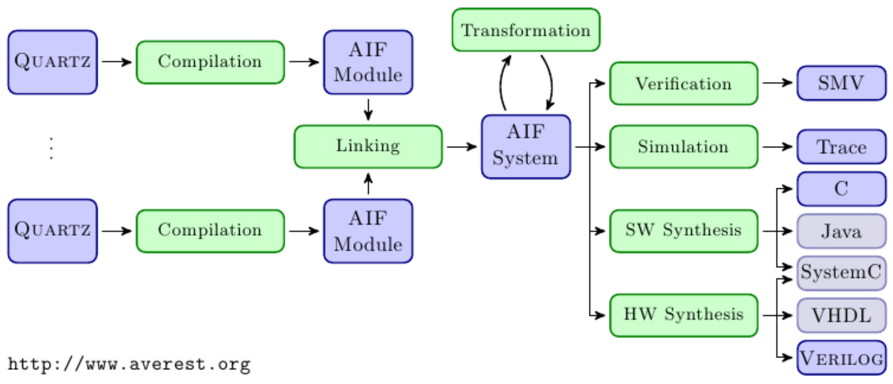


Fig. 2.14: Design Flow of the Averest Framework

2.5 Related Work

This work will extend the Averest system by interactive verification techniques by embedding the AIFProver. Hence synchronous guarded actions are used as a system description. The use of guarded actions in general, and even of synchronous guarded actions in particular, is not new, neither for theorem proving nor for interactive verification or even system descriptions.

In general, guarded actions are well suited to describe systems with different MoCs [BGSS12, BrSc11, GeSc13c]. Different MoCs are thereby obtained by different ways to select actions for execution.

It seems to be impossible to use tools for one of these MoCs for guarded actions that are based on another MoC. In particular, most of the rewrite-based verification tools and theorem provers, like Unity [ChMi89], SAL [MORR04], Rodin [Abri10], and Murphi [Dill96] are based on interleaved guarded actions, but the representation for synchronous systems uses SGAs.

The guarded actions have been suggested by Dijkstra in [Dijk75] as a formalism to reason about programs and have been used by Alur and Henzingers' reactive modules [AlHe99]. In this work, we propose the use of *SGAs* as a basis for interactive verification of synchronous programs.

Concerning hardware verification, Staunstrup and Greenstreet already proposed the use of their 'Synchronized Transitions' [StGr88, Stau97] in 1988 as a basis for efficient verification of hardware designs. The differences between *SGAs* and Hoare's parallel commands [Hoar78] are that *SGAs* do not have a disjoint set of variables and they communicate over shared variables (broadcast) instead of synchronous message passing. Using the Larch Prover, a rewriting-based theorem prover for the Larch language [GaGu89], they were able to verify some non-trivial hardware designs in a series of papers [StGG90, StGG92, StMe95, MeSt95, Stau97]. In contrast to our *SGAs*, Staunstrup and Greenstreets' synchronized transitions are asynchronous in the sense that only a subset of the enabled actions is selected for execution. In addition to the different models of computation, they used the guarded actions primarily as system descriptions which we think is unreadable for large systems. Thus, we consider guarded actions as an intermediate representation obtained by compilation from a better readable Quartz program.

Darringer [Darr79] proposed in 1979 the use of program verification techniques for the verification of hardware designs. In particular, he focused on Floyd's inductive assertions [Floy67] and symbolic simulation. [ZhHo92] presents an algorithm to translate transistor-level circuits to programs to apply Hoare-style verification to the verification of switching circuits.

Recent work by Mike Gemünde et al [TBGS14, BGSS13, Gemu13a, BGSS12] extended the SGAs to clocked guarded action that allow to refine a reaction step by sub-clocks and represent languages/models like Signal and Concurrent Action Oriented Specifications (CAOS) [HoAr04].

There are many approaches transforming *SGAs* to other languages or MoCs: In [BrGS10], an automatic translation to SystemC by generating a dynamic schedule for the modules in order to preserve the semantics was presented; and in [BaBS11b] *SGAs* are translated to asynchronous dataflow process networks.

The approach in [ScBr08, BrSc08a] refines the translation of *SGAs* to transition systems at the level of micro steps. The intention of [ScBr08, BrSc08a] was to use these transition systems at the micro step level to perform causality analysis by means of theorem proving and bounded model checking. In [BrSc11] a representation of SGAs by interleved guarded actions (IGAs) for causality analyses was presented that in fact is the same translation we will introduce, but the approach presented here also translates the temporal logic specification and defines some improvements for implementation.

Clearly, the execution of the enabled *SGAs* in a step follows implicitly a causal ordering to define a deterministic result. Similarly, confluence must be ensured to guarantee deterministic results for *IGAs*.

In contrast to Dijkstra's guarded commands [Dijk75], *IGAs* have only a single *repetitive construct* consisting of the entire set of *IGAs*. Hence, our translation targets a subset of Dijkstra's guarded commands.

2.6 Specifications

Floyd founded the idea of formal methods and formal verification for computer programs [Floy67]. He wanted to invent a *rigorous standard* for proofs about them. Hoare realized this by inventing the Hoare calculus that formally defines the Boolean conditions satisfied by a program part in case certain preconditions are met before execution. Nowadays, temporal logics [ClES83, EMSS91, Emer90] are used for this purpose. Besides others, the Computational Tree Logic (CTL) [EmCl80] and the Linear Temporal Logic (LTL) [Pnue77a, Emer90] are important representatives. In this work we will rely on a suitable property specification language for verification that will be LTL, since it is well-known that branching time logics like CTL do not lend themselves well for modular verification [KuVa95].

2.6.1 Pre- and Postconditions

Pre- and postconditions as well as assumption and assertions are used to define safety properties. These properties are usually Boolean conditions describing a state. They can be easily checked, but the expressiveness is rather weak (equivalent to safety properties).

2.6.2 Syntax and Semantics of Linear Temporal Logic (LTL)

LTL is a popular formalism for the specification of temporal properties.

Definition 8 (Linear Temporal Logic). *For a given set of Boolean variables (propositions) $\mathcal{V}$, we define the set of* LTL *formulas by the following grammar:*

$$\begin{array}{rcl}
\mathsf{LTL} & := & \mathsf{A}\varphi \\
\varphi & := & \mathcal{V} \\
& | & \neg\varphi \\
& | & \varphi \vee \varphi \\
& | & \mathsf{X}\varphi \\
& | & [\varphi\, \underline{\mathsf{U}}\, \varphi] \\
& | & \overleftarrow{\mathsf{X}}\varphi \\
& | & \overleftarrow{\underline{\mathsf{X}}}\varphi \\
& | & [\varphi\, \overleftarrow{\underline{\mathsf{U}}}\, \varphi].
\end{array}$$

The set LTL contains state formulas and the formulas generated from φ represent path formulas.
It is well-known that these operators are sufficient to define LTL, but for convenience, we may also introduce further operators:

Definition 9.

$$\begin{array}{lcll}
\mathsf{G}\varphi & := & [\varphi\, \mathsf{U}\, 0] & \textit{(always)} \\
\mathsf{F}\varphi & := & [1\, \underline{\mathsf{U}}\, \varphi] & \textit{(eventual)} \\
[\varphi\, \mathsf{U}\, \psi] & := & [\varphi\, \underline{\mathsf{U}}\, \psi] \vee \mathsf{G}\varphi & \textit{(weak until)} \\
[\varphi\, \mathsf{W}\, \psi] & := & [\neg\psi\, \mathsf{U}\, (\varphi \wedge \psi)] & \textit{(weak when)} \\
[\varphi\, \underline{\mathsf{W}}\, \psi] & := & [\neg\psi\, \underline{\mathsf{U}}\, (\varphi \wedge \psi)] & \textit{(strong when)} \\
[\varphi\, \overleftarrow{\mathsf{B}}\, \psi] & := & \neg[\neg\varphi\, \overleftarrow{\underline{\mathsf{U}}}\, \psi] & \textit{(past weak before)} \\
[\varphi\, \overleftarrow{\underline{\mathsf{B}}}\, \psi] & := & \neg[\neg\psi\, \overleftarrow{\underline{\mathsf{U}}}\, \varphi \wedge \neg\psi] & \textit{(past strong before)}
\end{array}$$

The semantics of LTL is usually given with respect to an infinite path through a transition system (a Kripke structure).

Definition 10 (Infinite Path). *An* infinite path *is a function $\pi : \mathbb{N} \to \mathcal{S}$ with $(\pi^{(t)}, \pi^{(t+1)}) \in \mathcal{R}$, where we denote the t-th state of the path π as $\pi^{(t-1)}$ for $t \in \mathbb{N}$, $t > 0$.*

The semantics of path formulas of a transition system $\mathcal{T}$ is defined by the relation $(\mathcal{T}, \pi, t) \models \varphi$ that defines if a path formula φ holds on position t of a path π of a transition system $\mathcal{T}$ (see e. g. [Schn03] for a full definition). These infinite paths are nothing else than infinite sequences of assignments to the variables $\mathcal{V}$. The semantics of LTL is typically defined as follows [Emer90, Schn03]:

Definition 11 (Semantics of LTL).

- $(\mathcal{T}, \pi, t) \models p$ *holds iff* $p \in \mathcal{L}(\pi^{(t)})$ *for every* $p \in \mathcal{V}$
- $(\mathcal{T}, \pi, t) \models \mathsf{X}\varphi$ *holds iff* $(\mathcal{T}, \pi, t+1) \models \varphi$
- $(\mathcal{T}, \pi, t) \models [\varphi\, \underline{\mathsf{U}}\, \psi]$ *holds iff there is a* δ *such that* $(\mathcal{T}, \pi, t+\delta) \models \psi$ *and for all* $x < \delta$, *we have* $(\mathcal{T}, \pi, t+x) \models \varphi$

- $(\mathcal{T},\pi,t) \models [\varphi\, \mathsf{U}\, \psi]$ *holds iff* $(\mathcal{T},\pi,t) \models [\varphi\, \underline{\mathsf{U}}\, \psi]$ *or for all* x*, we have* $(\mathcal{T},\pi,t+x) \models \varphi$.
- $(\mathcal{T},\pi,t) \models \overleftarrow{\mathsf{X}}\varphi$ *holds iff* $t=0 \vee (\mathcal{T},\pi,t-1) \models \varphi$
- $(\mathcal{T},\pi,t) \models \overleftarrow{\underline{\mathsf{X}}}\varphi$ *holds iff* $t>0 \wedge (\mathcal{T},\pi,t-1) \models \varphi$
- $(\mathcal{T},\pi,t) \models [\varphi\, \overleftarrow{\underline{\mathsf{U}}}\, \psi]$ *holds iff there is a* δ *such that* $\delta \leq t$,$(\mathcal{T},\pi,\delta) \models \psi$ *and for all* x *with* $\delta < x \leq t$*, we have* $(\mathcal{T},\pi,x) \models \varphi$

$\mathsf{A}\varphi$ *holds in a state* s *of* $\mathcal{T}$ *if all infinite paths* π *starting in* s *satisfy* $(\mathcal{T},\pi,0) \models \varphi$*. Finally, a transition system* $\mathcal{T}$ *satisfies a LTL formula* $\mathsf{A}\Phi$ *if all initial states satisfy* Φ*, in this case, we write* $\mathcal{T} \models \mathsf{A}\Phi$*.*

In [ChMP92, Schn01b, Schn03, MoGS12], a temporal logic hierarchy has been defined in analogy to the hierarchy of ω-automata. Following [Schn01b], we define the hierarchy of temporal formulas by the grammar rules of Figure 2.15:

$P_{\mathsf{G}} ::= \mathcal{V} \mid \neg P_{\mathsf{F}} \mid P_{\mathsf{G}} \wedge P_{\mathsf{G}} \mid P_{\mathsf{G}} \vee P_{\mathsf{G}}$ $\mid \mathsf{X} P_{\mathsf{G}} \mid [P_{\mathsf{G}}\, \mathsf{U}\, P_{\mathsf{G}}] \mid \overleftarrow{\mathsf{X}}\, P_{\mathsf{G}} \mid [P_{\mathsf{G}}\, \overleftarrow{\mathsf{U}}\, P_{\mathsf{G}}]$	$P_{\mathsf{F}} ::= \mathcal{V} \mid \neg P_{\mathsf{G}} \mid P_{\mathsf{F}} \wedge P_{\mathsf{F}} \mid P_{\mathsf{F}} \vee P_{\mathsf{F}}$ $\mid \mathsf{X} P_{\mathsf{F}} \mid [P_{\mathsf{F}}\, \underline{\mathsf{U}}\, P_{\mathsf{F}}] \mid \overleftarrow{\mathsf{X}}\, P_{\mathsf{F}} \mid [P_{\mathsf{F}}\, \overleftarrow{\underline{\mathsf{U}}}\, P_{\mathsf{F}}]$
$P_{\mathsf{Prefix}} ::= P_{\mathsf{G}} \mid P_{\mathsf{F}} \mid \neg P_{\mathsf{Prefix}} \mid P_{\mathsf{Prefix}} \wedge P_{\mathsf{Prefix}} \mid P_{\mathsf{Prefix}} \vee P_{\mathsf{Prefix}}$	
$P_{\mathsf{GF}} ::= P_{\mathsf{Prefix}}$ $\mid \neg P_{\mathsf{FG}} \mid P_{\mathsf{GF}} \wedge P_{\mathsf{GF}} \mid P_{\mathsf{GF}} \vee P_{\mathsf{GF}}$ $\mid \mathsf{X} P_{\mathsf{GF}} \mid [P_{\mathsf{GF}}\, \mathsf{U}\, P_{\mathsf{GF}}] \mid [P_{\mathsf{GF}}\, \underline{\mathsf{U}}\, P_{\mathsf{F}}]$ $\mid \overleftarrow{\mathsf{X}}\, P_{\mathsf{GF}} \mid [P_{\mathsf{GF}}\, \overleftarrow{\mathsf{U}}\, P_{\mathsf{GF}}] \mid [P_{\mathsf{GF}}\, \overleftarrow{\underline{\mathsf{U}}}\, P_{\mathsf{F}}]$	$P_{\mathsf{FG}} ::= P_{\mathsf{Prefix}}$ $\mid \neg P_{\mathsf{GF}} \mid P_{\mathsf{FG}} \wedge P_{\mathsf{FG}} \mid P_{\mathsf{FG}} \vee P_{\mathsf{FG}}$ $\mid \mathsf{X} P_{\mathsf{FG}} \mid [P_{\mathsf{FG}}\, \underline{\mathsf{U}}\, P_{\mathsf{FG}}] \mid [P_{\mathsf{G}}\, \mathsf{U}\, P_{\mathsf{FG}}]$ $\mid \overleftarrow{\mathsf{X}}\, P_{\mathsf{FG}} \mid [P_{\mathsf{FG}}\, \overleftarrow{\underline{\mathsf{U}}}\, P_{\mathsf{FG}}] \mid [P_{\mathsf{G}}\, \overleftarrow{\mathsf{U}}\, P_{\mathsf{FG}}]$
$P_{\mathsf{Streett}} ::= P_{\mathsf{GF}} \mid P_{\mathsf{FG}} \mid \neg P_{\mathsf{Streett}} \mid P_{\mathsf{Streett}} \wedge P_{\mathsf{Streett}} \mid P_{\mathsf{Streett}} \vee P_{\mathsf{Streett}}$	

Fig. 2.15: Classes of the Temporal Logic Hierarchy

Definition 12 (Temporal Logic Classes). *We define the logics* TL_κ *for* $\kappa \in \{\mathsf{G}, \mathsf{F}, \mathsf{Prefix}, \mathsf{FG}, \mathsf{GF}, \mathsf{Streett}\}$ *by the grammar rules given in Figure 2.15, where* TL_κ *is the set of formulas that can be derived from the non-terminal* P_κ *(*$\mathcal{V}$ *represents any variable* $v \in \mathcal{V}$*).*

TL_{G} is the set of formulas where each occurrence of a weak/strong temporal operator is positive/negative, and similarly, each occurrence of a weak/strong temporal operator in TL_{F} is negative/positive. Hence, both logics are dual to each other, which means that one contains the negations of the other one. $\mathsf{TL}_{\mathsf{Prefix}}$ is the Boolean closure of TL_{G} and TL_{F}. The logics $\mathsf{TL}_{\mathsf{GF}}$ and $\mathsf{TL}_{\mathsf{FG}}$ are constructed in the same way as TL_{G} and TL_{F}; however, there are two differences: (1) these logics allow occurrences of $\mathsf{TL}_{\mathsf{Prefix}}$ where otherwise variables would have been required in TL_{G} and TL_{F}, and (2) there are additional ‘asymmetric’ grammar rules. It can be easily proved that $\mathsf{TL}_{\mathsf{GF}}$ and $\mathsf{TL}_{\mathsf{FG}}$ are also dual to each other, and their intersection strictly contains $\mathsf{TL}_{\mathsf{Prefix}}$. Finally, $\mathsf{TL}_{\mathsf{Streett}}$ is the Boolean closure of $\mathsf{TL}_{\mathsf{GF}}$ and $\mathsf{TL}_{\mathsf{FG}}$. While there are syntactic restrictions on $\mathsf{TL}_{\mathsf{Streett}}$, i. e. not every LTL formula is a $\mathsf{TL}_{\mathsf{Streett}}$ formula, $\mathsf{TL}_{\mathsf{Streett}}$ contains for each LTL formula an equivalent formula, and nearly all formulas used in practice belong to $\mathsf{TL}_{\mathsf{Streett}}$ [MoSL08]. Moreover, for those formulas not in $\mathsf{TL}_{\mathsf{Streett}}$, it is typically not difficult to find an equivalent one in $\mathsf{TL}_{\mathsf{Streett}}$.

2.7 Verification Techniques

In the following some verification techniques are described. Especially, the work of Manna and Pnueli must be mentioned [MaPn95b, MaPn92]. They describe the formal verification of reactive systems with safety specifications.

There exist many verification techniques. Model checking is widely used, because it is fully automatic. Many improvements are already presented in the literature to avoid or bypass the implied disadvantages. Techniques and tools before the advent of model checking are theorem proving and interactive verification. These techniques are originated from Floyd's vision and Hoare's realization of the Hoare calculus.

2.7.1 Model Checking

Model checking [ClGP99] is without any doubt one of the success stories of modern computer science [GrVe08]. Especially, the invention of Symbolic Model Checking [McMi92a, BCMD92, McMi93a, MoPS11] based on Binary Decision Diagrams (BDDs) [Aker78, Brya86] allowed to verify large systems [BCMD92, KuLa93, FGHH05]. However, despite of the tremendous progress in research made during the last two decades, model checking still suffers from the state-space-explosion problem, which means that in the worst case, the runtime for model checking grows exponentially with the size of the system to be verified. In practice, this leads to enormous requirements for verifying large systems. For this reason, model checking – even in its restricted versions of bounded [MoRS03, BCCS03a, LBHJ04] and SAT-based model checking [Brad11, BCCZ99, McMi03, ShSS00, McMi03a] – will in general not be applicable to large systems. Some approaches try to tackle the problem with computing power by distributing the verification task to a cluster, e.g. parallel verification [BCSVZ08, BaBCR10, BeCK10], but in general other techniques to reduce the complexity of a verification task is needed. In the literature many creative approaches are trackable to encounter the complexity, like *Runtime Verification* [MoGS12, FaFM10, FaFM10, BaLS10] where a specification is monitored during runtime and *Program Slicing* [Weis79, XQZW05, CFRR99, BiGa96], which decomposes program statements relevant to a certain slicing criterion that refers to some point(s) of interest. Besides various abstraction [CGJL03, HJMM04, ChJa12] and reduction techniques [Long93, Gode95, CEFJ96, EmSi96, ChJa12], inductive [CiGr12, Brad11, HaBS12, RaSS95] or compositional verification [McQS00, McMi98, McMi99a, BeCC98] is a key to fight the complexity given by large systems.

2.7.2 Theorem Proving

Interactive theorem proving has been considered as an interesting alternative since the beginning of formal verification [Gupt92, KeGr99]. Most of the proposed theorem provers are based on *Higher Order Logic (HOL)* like Isabelle [Paul94], PVS [OwRS92], VAMPIRE [KoVo13], and HOL [Gord86]. They usually follow the Logic of Computable Functions (LCF) style [GoMW79] of theorem provers, i.e., a functional programming language like ML is used as a meta language. Formulas, proof goals, and theorems are implemented as data types in ML, and proof rules are implemented as ML functions mapping existing theorems to new theorems. Since most functional languages offer interactive sessions, one can directly use these for interactive proof construction.

LCF style theorem provers [GoMW79] are sound, because they use only a very small set of deduction rules and axioms. For example, HOL [Gord86] relies on five axioms and eight basic deduction rules that are directly implemented on the data structure of theorems. These axioms and basic proof rules refer to the polymorphic λ-calculus that has been chosen as a foundation of HOL's logic. All other data types like natural numbers, lists, etc., and operations on these types like conjunction and addition are implemented as abbreviations of expressions of the λ-calculus. For these derived types, hundreds of more convenient proof rules have been implemented and proven on top of the primitive rules.

While this approach makes the theorem provers trustworthy, there is also a price to pay: A huge amount of interaction is often required for a proof; especially adding new data types to specify a proof goal is a complicated process. First of all, one has to implement the data type in terms of existing ones. Additionally, a proof of its non emptiness and the implementation of useful proof rules for it is required. The amount of work that is required for such an embedding is still very large even though there is some limited form of automation [Melh89a]. A promising approach is therefore the reduction of interaction by adapting the theorem provers to specific problems. Model checking and other decision procedures can thereby be integrated as particular proof rules [ScHo99, Gord00]. Although there was some early progress for register-transfer level hardware circuits, e.g., [KuSK93a], and general reactive systems as e.g., the STeP prover [BBCF00], there was little progress on automation for hardware descriptions at higher abstraction levels [Gord95].

2.7.3 Interactive Verification for High Level Description Languages

Since the system to be verified is typically given in a system description language, one has to embed that language in the theorem prover as well to formulate the verification problem. Several authors considered the embedding of different languages like VHDL [Reet95], ELLA [Boul92a], or Verilog [Gord95, Gord98]. However, often only small fragments of the languages were embedded and the verification was never supported by convenient proof rules. Embedding a non-trivial system description language with suitable proof rules requires many years of work. Thus, it is important to make sure that the right system representation together with a convenient set of proof rules is determined before these are embedded in an existing theorem prover. This thesis will determine the structure of the proof goals and rules such that a forthcoming work is able to completely embed the approach into a theorem prover.

2.7.4 Hoare Calculus

As the system descriptions given by synchronous languages are programs, it is natural to try to integrate classic software verification methods with the already established model checking techniques.

It is well-known that axiomatic semantics as given by the Hoare calculus [Floy67, Hoar69, Grie81, Apt81] are a basic foundation for the formal verification of software. This calculus provides proof rules for each kind of statement to reflect its axiomatic semantics.

Figure 2.16 shows Hoare's axioms for a simple sequential programming language. The verification of algorithms can even be done in a parameterized way that abstracts from the size of data structures (like array sizes) as well as abstracting from the data types

$$\textsf{nothing}: \quad \frac{}{\{\Phi\}\ \texttt{nothing}\ \{\Phi\}}$$

$$\textsf{assign}: \quad \frac{}{\{[\Phi]_x^\tau\}\ x = \tau\ \{\Phi\}}$$

$$\textsf{sequence}: \quad \frac{\{\Phi_1\}\ S_1\ \{\Phi_2\} \qquad \{\Phi_2\}\ S_2\ \{\Phi_3\}}{\{\Phi_1\}\ S_1; S_2\ \{\Phi_3\}}$$

$$\textsf{conditional}: \quad \frac{\{\sigma \wedge \Phi\}\ S_1\ \{\Psi\} \qquad \{\neg\sigma \wedge \Phi\}\ S_2\ \{\Psi\}}{\{\Phi\}\ \texttt{if}(\sigma)\ S_1\ \texttt{else}\ S_2\ \{\Psi\}}$$

$$\textsf{loop}: \quad \frac{\{\sigma \wedge \Phi\}\ S\ \{\Phi\}}{\{\Phi\}\ \texttt{while}(\sigma)\ S\ \{\neg\sigma \wedge \Phi\}}$$

$$\textsf{weaken}: \quad \frac{\models \Phi_1 \to \Phi_2 \qquad \{\Phi_2\}\ S\ \{\Phi_3\} \qquad \models \Phi_3 \to \Phi_4}{\{\Phi_1\}\ S\ \{\Phi_4\}}$$

Fig. 2.16: Hoare Calculus for a Sequential Programming Language

themselves (by considering polymorphic types). While systems with complex data flows are usually difficult to handle using model checking, they can often be easily verified by means of interactive verification using the Hoare calculus. Conversely, model checking lends itself better to control-intensive applications, since such systems often require the enumeration of all reachable states also in an interactive verification. An integration of model checking and interactive verification based on Hoare calculus or similar methods is therefore highly desired [JoSe94, KeGr99, ScKr97a].

Being used in interactive verification, these rules allow one to reduce a proof goal of that statement by generating corresponding proof goals for its sub-statements, so that a compositional or modular verification is obtained. Some of these rules require additional information: In particular, the rules for loops require invariant conditions as additional information that typically has to be provided by the user. While this might appear as a serious disadvantage of these approaches, it is the key to exploit the knowledge of the user about her/his system description that does not rely on a brute-force approach like model checking. For synchronous languages, this seems to be not too difficult since (1) model checking is already available and (2) the rules of the Hoare calculus 'only' have to be adapted to the considered synchronous language.

Indeed, there are already many approaches to the verification of concurrent programs by means of Hoare calculi as e.g. those using synchronous message passing to avoid the use of shared variables [BoHR97, Ding00, CCGO04, RBHH01, ApOl97] and others making use of critical region constructs [Broo85, RBHH01, ApOl97] for the same purpose. A good overview on many papers in this area is given in [Andr81a]. Again, all of them consider coarse grained systems consisting of concurrent sequential programs like in Hoare's famous CSP [LaSc84].

All of them have in common that their underlying MoC is different to the synchronous MoC, hence a usage without further modification is not possible. This work will show how to use or at least how to adapt techniques of the sequential MoC to be used to analyze systems described in the synchronous MoC.

Lamport proposed in [LaSc84, Lamp80a] to reason about safety and liveness conditions instead of pre- and post-conditions of the Hoare calculus. Although we are also interested in the temporal behavior of the synchronous programs, we aim at using the interactive verification for the data intensive parts only that be dealt well with the pre-/post-condition approach. Hence, we still make use of pre- and post conditions as given in the original Hoare calculus.

2.8 Adapting Interactive Verification to Synchronous Languages

However, none of these approaches can be directly used for the verification of synchronous programs. For this reason, we have to define a specialized Hoare calculus or similar interactive verification techniques for the synchronous model of computation. However, this adaptation is not straightforward: The main problem of this is that the assignment rule of the classic Hoare calculus implicitly defines a sequential programming model which is not sufficient for synchronous languages. Instead, the synchronous model of computation divides the execution of a program into macro steps that in turn consist of a finite number of micro steps (like assignments). The micro steps of a macro step are executed within the same variable environment, while all updates to the variables' values are synchronously performed at the level of macro steps. Due to complex control flow statements like pre-emption and parallel statements, these micro step actions of a macro step might be distributed over a large part of a program.

In [StCO06], a program transformation has been considered where parallel assignments are considered as an abstraction of sequential programs to save states for model checking. As we already start from a synchronous program, we have no need to determine which assignments should be clustered into one parallel assignment, since this is determined by the semantics of the synchronous programming language. However, a related problem is considered here: This work defines a program transformation to collect the actions of different threads into a tuple action so that the concurrency is only available in these synchronous tuple-assignments. Such tuple-assignments have already been considered by Martin and Tucker [MaTu89] who used these assignments to reason about the correctness of systolic algorithms. Indeed, they already introduced the tuple-assignment form that will be used in Chapter 3 to employ for synchronous programs for the approach. Since synchronous languages did not yet exist when [MaTu89] was published, they did not discuss program transformations into that normal form, and directly used such descriptions without discussing their origin.

2.9 Modular Verification

Since reactive systems are often used in safety-critical applications, their functional correctness is of essential importance. For this reason, simulation and formal verification are routine steps in their design flows, and in particular, model checking is often used for this purpose. However, due to the well-known state space explosion problem, a modular or compositional verification [Roev98, BoRo98] is desired where modules can be replaced by their already verified properties. Large reactive systems can only be verified by modular or compositional approaches despite the tremendous progress on model checking procedures we have seen in the past two decades. Another reason for modular verification is that modules are defined for being reused later on, and therefore the effort for formal verification amortizes when one can simply reuse also the already verified properties.

Recent work of Schneider and Brandt [Schn09, BrSc09] describes a method to compile single synchronous modules on their own to *SGAs* that can be later linked together. Modular compilation was not possible for (imperative) synchronous languages, and since modular compilation is now available, it is natural to also establish modular verification. Clearly, it has to follow the semantics of modules and module calls, and this is defined differently for synchronous languages: While functional languages like Lustre consider modules as functions without side effects and essentially assume that all modules are started in parallel to all other modules, languages like Esterel and Quartz allow module calls in arbitrary statements. In these languages, modules are declared with input and output parameters (variables) so that the body statement of the module is only allowed to read its input variables and to write to its output variables. If the module is later on called in a context module, the input parameters are replaced with expressions of the same type, output parameters are replaced with local or output variables of the calling module, and thus, the assignments of the called modules then become assignments of the calling module. The calling module may also make assignments to its local and output variables, so that the two behaviors are combined. Moreover, module calls can be delayed to an arbitrary point of time, and a running module may be aborted or suspended by the context of its calling module.

Also, the simulation of synchronous modules has to be finally aborted so that temporal logic specifications referring to infinite behaviors cannot be completely answered. The results presented in this work are not only useful for modular verification: in [ABKV03], the authors considered the problem to make specifications for the simulation of reactive systems, which is difficult since the simulation has to be aborted after some finite time, so that properties that refer to the infinite behavior of the system cannot be completely answered. The results can be also used for simulation in pre-emption contexts or to verify temporal logic specification with bounded model checking or runtime verification [MoGS12].

Another way to deal with this problem is the use of *compositional or modular verification* [ClLM89a, GrLo91, HaLR93, AAHM99, LaGr98, RBHH01], where one first verifies some properties for single modules without their later context, and then makes later on use of these properties when the entire system is verified. Clearly, compositional verification is not new; many different approaches have already been developed including the assume-guarantee, assumption-commitment or rely-guarantee styles of compositional reasoning (see [RBHH01]). Moreover, the distinction between open and closed systems leads to the notion of *module checking* [KuVa96, KuVW01, Gode03] where one considers properties that hold even if the environment later on restricts input traces to some subset.

Concerning related work on modular verification of Esterel programs, I found only one paper of Merceron [Merc96] as related work. In contrast to this work, her paper does not consider the translation of synchronous programs to transition systems, and only considers the application of existing preservation results to the modular verification, which do neither consider substitution of parameters, nor delayed starting points, nor abortion/suspension.

Other papers like [HaLR93] consider the decomposition of a global specification φ of a parallel composition $M_1 \parallel M_2$ to local specifications φ_1 and φ_2 such that $M_1 \models \varphi_1$ and

$M_2 \models \varphi_2$ implies $M_1 \parallel M_2 \models \varphi$. This is a very important problem that is however not related to the here encountered problems: it is assumed that a specification φ_i of a module M_i has already been verified, but now the module is called in a general context (not just parallel composition) where parameters are substituted by expressions, and where the module's behavior is completed by assignments of the context. Thus, the problem is to determine preservation results for the verified specification φ_i of a module M_i and there is no need to decompose φ to the φ_is.

In [CBKT13] a symbolic decomposition of a transition relation by a locality check using decision diagrams is presented. They work directly on the low level description of a transition relation, this work instead uses synchronous guarded actions as description.

Outline

In the following Chapter, the definition of a Hoare calculus for Quartz on the source-code level is discussed. It is shown that a definition of a Hoare-like rule for every statement is cumbersome and that a normal form that collects the behavior of a macro step at a single program part allows easily to define such a calculus. Unfortunately, the main result of this chapter will be the impossibility of translating every Quartz program to this normal form. Hence, an other approach is presented in Chapter 4 and extended by the following chapters.

Chapter 3

Interactive Verification of Synchronous Programs at Source-Code Level

In this chapter, the definition of a Hoare calculus at source-code level of Quartz is discussed. Introducing a Hoare calculus seems to be an easy task at the first glance: the rules only need to be extended for the synchronous statements, and even concurrent execution was considered for Hoare rules in the past [OwGr76a, OwGr76b, ApOl97, RBHH01]. The SOS rules defining the semantics of Quartz are a good starting point for that. Firstly, the idea of defining such rules is sketched. However, it will be explained in Section 3.1 that this is not at all straightforward. The main problem presented in Section 3.1.1 is thereby the presence of statements for concurrency in a form that has not yet been considered for Hoare calculus.

Having identified these problems, an alternative approach consisting of a normal form for Quartz programs and Hoare rules for these programs is shown in Section 3.2. The usefulness of these method is discussed in Section 3.2.4 by verifying an parallel algorithm. The main result of this chapter is presented in Section 3.2.5 that proves that the translation to the normal form is not possible without adding additional variables to the programs in general. The reason therefore is that there are some combinations of statements that introduce problems similar to the goto statement in sequential programs [Dijk68, KoFu06]. Besides the general transformation, two transformations for restricted classes of synchronous programs are given in Section 3.2.6 where it is not necessary to introduce the additional variables. These transformations led to the approach presented in Chapter 4.

3.1 Adapting the Hoare Calculus for Quartz Programs

This section sketches the basic idea of using the SOS rules to define the Hoare rules. In sequential programming languages, each statement updates a global state in the order of the execution of the statements. This allows a compositional approach/definition of the Hoare rules for these languages. In the synchronous MoC, several statements are executed synchronously in a macro step and all influence the execution independently of the order in which they occur in the source code, as a result of the data-dependent execution order. Hence, the idea of defining Hoare rules for Quartz is based on collecting the behavior of a macro step (all micro-step actions) and identifying the end of a macro step first. Then, reasoning about the whole macro step and the execution of delayed assignments is done.

This results in the requirement of two different rule sets for instantaneous statements and composed statements.

The synchronous MoC requires that the variable environment does not change during execution of micro steps, and synchronous updates of all variables have to be performed at the beginning of a macro step. The assignment rule of Hoare calculus immediately updates variables after each assignment, which is an encoding of the sequential MoC that is not compatible with the synchronous MoC. Instead of having a pre-condition before executing a statement and a post-condition that is fulfilled after the execution of that statement, it is required that the pre-condition holds in the first macro step of a statement and the post condition holds in the last macro step of that statement. Hence, for instantaneous statements, both points of time are the same and there exists just a single satisfied condition and only a composed statement that consumes time has a pre- and post-condition. This directly reflects the difference of the two MoCs.

The ideas for the rules are presented in Figure 3.1. The precondition φ holds in the first step of the statement S and the post-condition Ψ holds in the last step of the statement S. Hence, these conditions are the same for instantaneous statements.

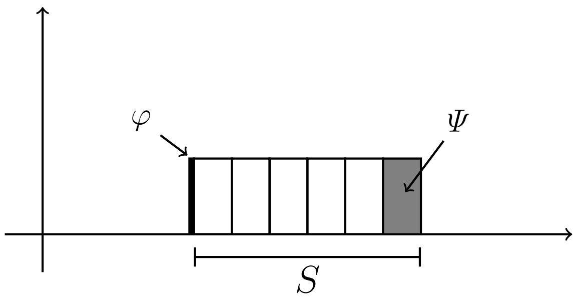


Fig. 3.1: Idea for Macro Step Rules

3.1.1 Difficulties to Define a Hoare Calculus for Quartz

The first attempts to define a Hoare calculus for synchronous languages made clear very early that difficult problems have to be solved, and the results given in this section show that a direct definition of a Hoare calculus at the level of macro steps is not easily possible:

- The variable environment does not change during execution of micro steps, and synchronous updates of all variables have to be performed at the beginning of a macro step. In contrast, the assignment rule of Hoare calculus immediately updates variables after each assignment, which is an encoding of the sequential execution of actions that is not compatible with synchronous concurrency.
- Several assignments may be triggered during the same macro step (e.g. by the parallel, abort and suspend statements), and these assignments may belong to different sub-statements. As the rules of Hoare calculi are compositional and follow the syntax tree of the program, this would require to traverse the entire syntax tree of the program

for every reaction step in order to reach out for enabled assignments. Hence, the local reasoning typically used in Hoare logic does no longer work, and must be replaced by a global view on the program.

- Also the consideration of the delayed assignments that are executed *logically after* the immediate assignments to consider their effects, requires a reiteration over the syntax tree.
- Synchronous programs read inputs in every reaction step, while sequential programs considered in Hoare calculus do typically not read inputs.
- Synchronous Quartz programs determine unique values for all output and local variables for every macro step. In case no assignment determines a value for a variable, a corresponding reaction to absence determines it.
- The execution of statements may or may not require time, depending on the value of inputs. Thus, the determination of the parts of a program that belong to the current macro step is again a proof task that has to be handled in Hoare calculus.
- Reincarnation of local variables is a very tricky problem that has to be handled in the Hoare calculus as well. To this end, variable environments have to be considered that keep track of the values of the different reincarnations of a variable that exist at the same time. While compilers know how to deal with schizophrenia, the very difficult work [ScBS06, Schn09] would have to be re-implemented entirely in the Hoare calculus rules.
- Causality problems have to be detected and solved (if possible) to avoid consistency problems or potential non-determinism in the potential Hoare rules. While causality problems are well-understood and compilers know how to handle them, the existing procedures would have to be re-implemented in a Hoare calculus as well.

3.2 Adapting the Hoare Calculus for Quartz in a Normal Form

As already described, it is not straightforward to define a Hoare calculus for synchronous languages. The translation to sequential programs for verification with the Hoare calculus has also serious drawbacks, as the control flow of the original program is completely destroyed by the translation. Hence, even though the classic Hoare calculus is applicable this way, the necessary user interaction, e.g. to provide additional information/invariants, becomes very inconvenient.

Instead of defining rules that represent the behavior of each statement that additionally have to collect micro step actions, an alternative approach that is based on a normal form is presented here. It was published in [GeSc12] and represents a compromise between the two mentioned approaches of translating to sequential programs and defining Hoare rules for each statement. The approach operates in two steps, the first stage transforms a given Quartz program to a normal form such that all micro step actions are collected into a single assignment statement. Thereby, the rest of the syntactic structure of the synchronous program should be retained. Programs obtained this way are called to be in *sequential synchronous tuple assignment (SSTA)* normal form. We show that in principle all Quartz programs can be automatically transformed into this form, but this transformation require additional variables, and even though a manual rewriting often produces better code for verification as a naive approach that follows the C-synthesis.

It is important to note that this normal form allows the preservation of most of the program's control structures, especially the loops, and therefore, it allows the use of their invariants for the verification with the Hoare calculus. This way, interactive verification based on a generalized Hoare calculus is conveniently used. One should also note that this normal form is not a canonical (unique) normal form.

The transformation solves most of the problems mentioned in Section 3.1.1. Since all assignments in a SSTA form program are collected at a single place of the program, iterations over the syntax tree to find the triggered actions of a macro step can be avoided. Additionally, write conflicts are easily detected and reincarnations of local variables are easily identified as well. The values of delayed assignments can be evaluated and will be assigned to the variables as soon as a **pause** statement is reached. We assume in the following that the considered programs have no causality conflicts, because causality is checked during compilation.

3.2.1 Definition of Sequential Synchronous Tuple-Assignment Form

The main idea of this section is based on the SSTA form that represents a synchronous program as a sequential program that makes use of synchronous tuple-assignments.

Definition 13 (Synchronous Tuple-Assignments (STAs)). *Given that x_i and y_i are pairwise different left-hand side expressions of the language Quartz and $x_1 = \tau_1$, ..., $x_m = \tau_m$ and* **next**$(y_1) = \pi_1$, ..., **next**$(y_m) = \pi_m$ *are assignments that are causally ordered such that there are no read-after-write conflicts, i.e. that τ_i only has occurrences of x_1,...,x_{i-1}, then we call the following statement a* synchronous tuple-assignment*:*

$$(x_1,\ldots,x_m).(y_1,\ldots,y_n) = (\tau_1,\ldots,\tau_m).(\pi_1,\ldots,\pi_n)$$

Using STAs, we are able to aggregate assignments $x_i = \tau_i$ and **next**$(y_i) = \pi_i$ that are executed in one macro step into a single tuple assignment. Thus, we wish to rewrite a given synchronous program such that in every macro step, at most one STA will be executed that defines the entire data flow of that macro step.

Position of the STA in a Reaction Step

The position of the STAs has a great impact on the required rules. Hence, it is important to determine the best position for them. An ideal position would be the direct coupling of them with a **pause** statement, because then only one additional rule is necessary.

Unfortunately, the example in Figure 3.2 shows that a direct couplings (STA then **pause**) is not possible without an unwanted rewriting of the program structure or additional variables:

- It is not possible to shift the assignment of c = 2 (Line 10); into the loop without adding an additional variable. Let us assume that we are in the loop in the last macro-step and the loop condition does not hold, hence we have to execute the assignment for c, but after execution of this assignment (and all other micro step actions of the current macro step) we reach the corresponding **pause** statement (as a result of the definition of the desired form), which finishes the current macro-step. Hence, we do not leave the loop in this step and in the next step the loop condition may hold again (since it is an input), which leads to a wrong computation.

```
1  while(i){
2      c=1;
3      l0: pause;
4      if(j) {
5          a=1; l1: pause; b=1;
6      } else {
7          a=2; l2: pause; b=2;
8      }
9  }
10 c=2;
11 l3: pause;
```

Fig. 3.2: Example to Determine the Position of a STA

- For the example in Figure 3.2 it is also impossible to shift the execution of the last macro-step behind the loop, because therefore the last step of the loop body must be computed, but the two branches (Lines 5 and 7) of the if-then-else statement are not distinguishable (without unwanted merging of control- and data-flow or additional variables) and so it is impossible to execute these micro-steps after leaving the loop.

Hence, without rearranging the whole program or adding additional variables it is impossible to get the desired form for this example. In the next section, we describe the arranging of micro-step actions into single execution units not directly but followed by a pause statement such that only a single STA is executed in each macro step. Hence, the normal form is defined as:

Definition 14 (SSTA Form). *A synchronous program is in* sequential synchronous tuple-assignment (SSTA) form *if all its actions are STAs and between the execution of two STAs at least one pause statement is executed.*

The SSTA from does not allow two STA be executed in the same macro step, hence it implicitly prohibit the parallel statement.

3.2.2 Extension of Hoare Calculus for Programs in SSTA-Form

After applying a SSTA transformation that will be introduced in Section 3.2.6 and in Section 3.2.7 the concurrency available in one macro step of the original synchronous program has been reduced to a single STA that synchronously updates all variables. For example, Figures 3.24, 3.25, and 3.26 show examples of original and transformed statements.

The remaining statements after applying the transformation are the usual statements of a sequential programming language, we can make use of the Hoare rules of Figure 2.16 except for the assignment rule. The assignment rule is replaced by a more general new rule for STAs that represents the formal semantics of a STA:

Definition 15 (Tuple-Assignment Rule).

$$\textsf{STAAssign}: \quad \frac{}{\{\left[\left[\ldots[\Phi]_{x_1}^{\tau_1}\ldots\right]_{x_n}^{\tau_n}\right]_{y_1',\ldots,y_n'}^{\pi_1,\ldots,\pi_n}\}\ (x_1,\ldots,x_m).(y_1,\ldots,y_n)=(\tau_1,\ldots,\tau_m).(\pi_1,\ldots,\pi_n)\ \{\Phi\}}$$

The meaning of the above rule is as follows: First of all, note that the effect of the immediate assignment $x_i = \tau_i$ is already seen by the evaluation of the other right hand side expressions τ_j with $j > i$ as well as for all expressions π_k. Moreover, the delayed assignments are encoded by a concurrent substitution so that the values of the y'_k do not have an impact on the evaluation of the right hand side expressions and they are not accessible until the next pause statement is reached.

Definition 16 (Pause Rule).

$$\mathsf{pause}: \quad \frac{}{\{\left[[\ldots\Phi\ldots]^{\tau_1\ldots\tau_n}_{i_1,\ldots i_n}\right]^{y'_1\ldots y'_n}_{y_1\ldots y_n}\}\ \mathbf{pause}\ \{\Phi\}}$$

The meaning of the pause statement is to define the end of the current macro step and the start of the next one. The corresponding Hoare calculus rule has to update the values of the input variables i_k and finally the values of all next assignments y'_k has to be assigned to the variables y_k. Our new rules are quite similar to the original assignment rule and indeed they contain the assignment rule as a special case.

SSTA-Form and Pre-emption Statements

It is possible to define Hoare calculus rules for the pre-emption statements so that also pre-emption statements will be preserved. It is however also possible to remove these statements by modifying either the actions inside a STA, the loop conditions, or by surrounding parts of the code with simple loop or if-then-else statements. All these steps are similar to the generation of guarded actions during compilation, where the pre-emption statements are removed, too.

```
weak suspend {
  (a,b).() = (1,1).();
  l1: pause;
  (b).(c) = (2).(3);
  l2: pause;
  ().(c) = ().(3);
} when(i);
```

Fig. 3.3: Before Elimination

```
(a,b).() = (1,1).();
do {
  l1: pause;
  (b).(c) = (2).(3);
} while(i);
do {
  l2: pause;
  ().(c) = ().(3);
} while(i);
```

Fig. 3.4: After Elimination

For instance, removing a *weak suspend* statement (see Figure 3.3) requires to embed all contained **pause** statements and the associated STA for that macro step in a loop with the suspend condition. In this way, the transformed program (see Figure 3.4) behaves like the original one, because the weak suspend statements demand that as long as the condition holds, the control flow is suspended while the actions of the current macro step are executed.

The additional loops are also no problem for the Hoare calculus, because their invariants are trivial.

In case of a strong suspend statement, only the **pause** statements must be surrounded by a loop (see Figure 3.5), because this statement demands that as long as the condition holds, the control flow is suspended and no actions are executed.

```
(a,b).() = (1,1).();
do {
  l1: pause;
} while(i);
(b).(c) = (2).(3);
do {
  l2: pause;
} while(i);
().(c) = ().(3);
```

Fig. 3.5: SSTA Quartz Program after Elimination of Strong Suspend

Abortion statements can be replaced by if-then-else statements and the modification of all loop and suspend conditions in the scope of an abort statement. In general, we conclude that the choice between removing or preserving the pre-emption statements seems not to be relevant for an interactive verification by means of the Hoare calculus.

3.2.3 Comparing a Simple and a Manual SSTA Transformation

It is not difficult to see that every synchronous program can be translated to an equivalent program in SSTA form by using additional boolean variables for the locations. This is in analogy to the translation of control flow to data flow by the elimination of goto statements [BoJa66, AsMa71, BrSt72, Elgo76, Mill75], and it is essentially what happens when compilers translate synchronous programs to intermediate code like guarded actions [Schn09, BrSc11]: compilers make use of the location variables to encode the control flow of the program by adding assignments to these location variables. These assignments are, however, nothing else but simulated goto statements. As goto statements are considered harmful for structured programming [Dijk68, Dijk72], *a reduction that works without these additional variables is desired.* Moreover, the introduction of these variables is often unnecessary and it makes the verification by the Hoare calculus more difficult [KoFu06]. In particular, these assignments complicate the required invariants that the user must provide during the verification. Hence, an automatic transformation of synchronous programs to STA form obtained by translating control flow to data flow using assignments to location variables is not desired. Before describing some automatic transformations, a manual transformation into STA form is used to show the feasibility of the approach that preserves the control flow statements for verification. Therefore, this manual transformation is compared to the approach used in the synthesis to C-code. In the following, this transformation is described in more detail. Moreover, a transformation to SSTA form that preserves most control statements induces also a new kind of code generation for sequential languages. It will be shown that the manual rewriting creates better code for verification than the naive transformation.

Unfortunately, Section 3.2.5 will prove that a transformation to SSTA form in general is impossible. Thus, some synchronous programs can not be transformed into SSTA form without adding new variables. Nevertheless, the Sections 3.2.6 and 3.2.7 will show that there exists transformations into the SSTA form for most Quartz programs.

Simple SSTA Transformation based on the Idea of the C-Synthesis

A simple transformation to STA form is based on the EFSMs generated by the compiler. The advantage of the EFSM is that the actions of the macro step are already collected in the states of the EFSM. Indeed, one may interpret a state of the EFSM as another representation of a STA. Hence, a transformation to a STA form can be obtained by representing the EFSM as a synchronous program. For example, one may just use a loop that switches to the current control flow state in each step. However, such a SSTA form program would again translate the control flow to the data flow by assignments of the location variables. This procedure is nothing new, because the synthesis to sequential code, e.g. C code, is done in a similar way. Therefore, this transformation suffers from the same problems as the original Hoare calculus applied to synthesized sequential code since all control flow structures have been removed. Instead, it is shown that the manual transformation to STA form is preferred, because here it is possible to preserve most of the original syntax.

Example

For a better understanding, we consider a small example. The Quartz module in Figure 3.6 is an implementation of an algorithm to compute Fibonacci numbers. The program receives a natural number `inp` as input, and computes the natural number `f` and the Boolean event rdy as output. In its first macro step, the program receives the index `inp` of the Fibonacci number to be computed. During the computation, the program assigns in every macro step one Fibonacci number to output `f`. The program signalizes the end of the computation by the event `rdy`. In that case, the output `f` should contain the desired value.

The corresponding EFSM is shown in Figure 3.7. The EFSM has three states, where State 0 is the initial state and State 2 is the final state. In each macro step, all actions of the current state are executed according to their data dependencies.

For example, the action $\texttt{n} \leq 0 \Rightarrow \texttt{f = 0}$ of State 0 states that if `n` is less than or equal to zero, then 0 is immediately assigned to variable `f` in this state. After the execution of all actions, the next state is determined by the transitions and their corresponding conditions. It is important to know that all conditions of a state's outgoing transitions are disjoint as a result of the deterministic behavior of synchronous programs.

The result of the above mentioned procedure to generate program code in SSTA form is shown in Figure 3.8. As can be seen, where the program has been completely rewritten. Unfortunately, the local loop is replaced by a loop that contains the whole program and the global loop would also cover all local loops (even nested) if there would be more than one loop. To use this program for verification with the Hoare rules for STA, an invariant for the loop is required, but invariants are usually manually given. The devloper, however, might not

```
module Fib(nat ?inp,f,event !rdy) {
    nat k,g,n;
    n = inp;
    if(n <= 0)
        f=0;
    else {
        k = 1;
        g = 0;
        f = 1;
        while(k != n) {
            next(g) = f;
            next(f) = f+g;
            next(k) = k+1;
            l: pause;
        }
    }
    emit(rdy);
}
```

Fig. 3.6: Computing Fibonacci Numbers in Quartz

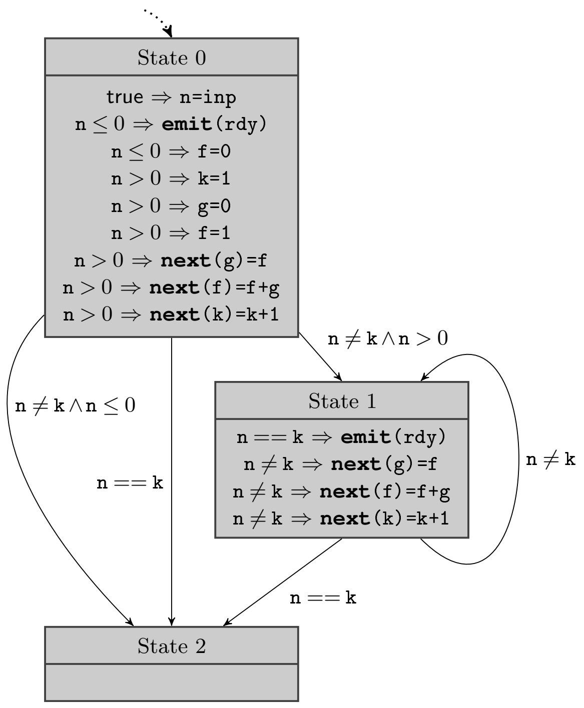


Fig. 3.7: EFSM of Module `Fib` (Figure 3.6)

```
module FSA(nat ?inp,f,event rdy){
  nat k,g,n,l;
  do {
    case
      (l==0) do
        (n,rdy,k,g,f).(g,f,k,l) =
          (inp,n<=0,1,0,(n>0?1:0)).
          (f,f+g,k+1,(n>0&n!=k?1:2));
      (l==1) do
        (rdy).(g,f,k,l) =
          (n==k).(f,f+g,k+1,(n!=k?1:2));
    default
      nothing;
    pause;
  } while (l!=2);
}
```

Fig. 3.8: Automatically Generated STA Code for Module `Fib`

have any clue for the global loop, but could give an invariant for the local loop. Even worse is the fact that the case statement contains a clause for each reachable control flow state.

```
module FSM(nat ?inp,f,event !rdy){
  nat k,g,n;
  if(n<=0)
    (n,f,rdy).() = (inp,0,true).();
  else {
    (n,k,g,f).(g,f,k,l) =
      (inp,1,0,1).(f,f+g,k+1,2);
    while(k!=n) {
      pause;
      if(k!=n)
        ().(g,f,k,l) = ().(f,f+g,k+1,2);
      else
        (rdy).() = (true).();
    }
  }
}
```

Fig. 3.9: Manually Transformed SSTA Code for Module `Fib`

In contrast, a manually generated SSTA code for module `Fib` is shown in Figure 3.9. Assuming that the code was correctly transformed by the user, this code has the advantage that only the actions are duplicated to the STAs to solve the problems discussed in Section 3.1.1 while the control flow has been completely preserved.

The verification of the automatically generated code using the Hoare calculus requires an invariant for the while loop that contains the whole program. In contrast, the verification of the manually translated code requires only an invariant for a loop comparable with the original loop. Clearly, the automatically generated code always contains only a single loop and therefore merges all loops of the given program into one which makes the use of loop invariants quite difficult.

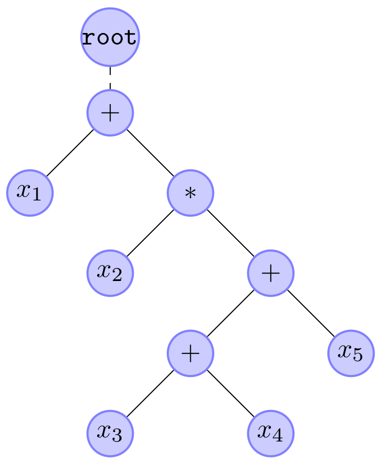
3.2.4 Experimental Results

The *manual rewrite* approach was applied to some classic sequential algorithms modeled in Quartz. Among other examples, an algorithm to compute Fibonacci numbers (see Section 3.2.3), the extended Euclidean algorithm, and Bubblesort were considered. These programs were transformed manually to SSTA form, and verified in the same way the sequential programs are verifiable without any problems. The SSTA form programs were not much longer than the original programs, and the already known classic proofs did work also for the synchronous programs. Hence, these examples substantiated the approach.

Of course, the advantage of the approach is that it is now possible to verify algorithms where several actions are executed in parallel. Therefore, the verification of the parallel algorithm for expression tree contraction was done.

Parallel Expression Tree Contraction

In [KaRa88, Meta97] a parallel algorithm for the evaluation of expression trees that runs in time $O(log(N))$ for expression trees with $O(N)$ leaf nodes is given. The input of this algorithm is an rooted expression tree, where the inner nodes correspond to expressions of addition (ADD) or multiplication (MUL) and the leaves correspond to constant values. During the evaluation of the expression, the tree is transformed and finally collapsed to a single node containing the result of the evaluation. The expression tree for the expression $(x_1 + (x_2 * ((x_3 + x_4) + x_5)))$ is given in the following, where x_i with $i \in \{1..5\}$ is a constant.



The Shunt Operation

The algorithm is based on tree contraction that is based on the *shunt* operation. The *shunt* operation is a pruning of a leaf followed by a shortcutting of the leaf's parent. In Figure 3.10, the *shunt* operation is applied to the leaf labelled with e.

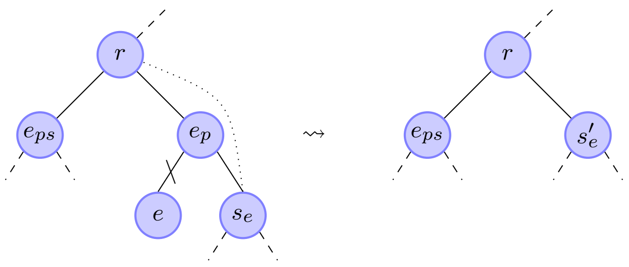


Fig. 3.10: *Shunt* Operation

The Idea of the Algorithm

The idea of the algorithm is to repeatedly apply *shunt* operations in parallel to as many nodes as possible in each iteration step. To circumvent conflicts between the shunt operations, the *shunt* operations are first applied to all odd numbered leaves that are the left children of their parents, and afterwards to the odd numbered leaves that are the right children of their parents. In that way, half of the leaves and the same number of inner nodes are removed in each iteration and all applied shunt operations do not interfere. In Figure 3.11, the pseudo code of this tree contraction algorithm is given (see [KaRa88]).

```
1) Label the leaves in order from left to right.
2) for ⌈log(n)⌉ operations do:
   a) Apply shunt in parallel to all odd numbered leaves
      that are the left children of their parents.
   b) Apply shunt in parallel to all odd numbered leaves
      that are the right children of their parents.
   c) Divide the indices of the leaf labels by two.
```

Fig. 3.11: Tree Contraction Algorithm

This algorithm was implemented in Quartz (see Figure 3.12) and transformed it into STA form manually (see Figure 3.13). Finally, the correctness of the implementation was proven.

Parallel Expression Tree Contraction in Quartz

In the implementation, expressions are stored in an array *expr* that holds tuples with components *opc*, *prt*, *sib*, *val*, *lft*, *a*, *b* and *bsy* with the following meanings:

- *opc* indicates either the root node (ROOT), a constant value (CST) or an operation (ADD or MUL)
- *prt* is the index of the parent node
- *sib* is the index of the sibling node
- *val* is the value of a leaf which must have op-code $opc \equiv \text{CST}$
- *lft*: if $opc \not\equiv \text{ROOT}$ holds, then this indicates whether it is a left child
- a,b are integers that specify the meaning of the expression so that if the expression is recursively evaluated to a value v, the value is $a \cdot v + b$
- *bsy* is a flag that indicates whether an array entry is in use

In addition to the expression tree, the algorithm also requires an array `leaves` that contains the indexes of the leaf nodes from left to right (for the above example the array `leaves` contain the indexes for the nodes x_1 to x_5 of the array representation for the expression tree). This array `leaves` is used to iterate the leaves of the expression tree.

Assumptions and Invariants

Since the initial values stored in the arrays are not restricted, it must be ensured that the represented data structure is really a tree and that the array `leaves` really refers to its leaf nodes. Therefore, we start with an initial assumption ensuring these consistency conditions, and prove that these properties remain valid during the execution of the algorithm:

Lemma 1. *The array Σ representing the expressions tree is a rooted, ordered binary tree with the following properties:*

- single root:
  $\exists r \in \Sigma.\, (r.opc \equiv \text{ROOT}) \wedge r.bsy \wedge \forall e \in \Sigma\ \neg e.bsy \vee (e.opc \not\equiv \text{ROOT})$
- binary tree:
  $\forall a, b \in \Sigma\ \mathit{with}\ (a = b.sib) \wedge a.bsy \wedge b.bsy\,.\, (a.prt = b.prt)$
  $\wedge \nexists c \in \Sigma\ (a \neq c \neq b) \wedge (a.prt = c.prt) \wedge c.bsy$
- siblings:
  $\forall e \in \Sigma\ \mathit{with}\ \neg((e.prt.opc \equiv \text{ROOT}) \wedge (e.opc \not\equiv \text{ROOT})) \wedge$
  $(e.opc \equiv \text{CST}) \wedge e.bsy\,.\, \exists! s.\, (s = e.sib) \wedge s.bsy$
- left/right:
  $\forall e \in \Sigma\ \mathit{with}\ \neg((e.prt.opc \equiv \text{ROOT}) \wedge (e.opc \not\equiv \text{ROOT})) \wedge$
  $(e.opc \not\equiv \text{ROOT}) \wedge e.bsy\,.\, a.lft \neq a.sib.lft$
- constants are leaves:
  $\forall a \in \Sigma\ \nexists b \in \Sigma\ (a.opc \equiv \text{CST}) \wedge (b.parent = a)$

The first property states that the expression tree has a single root node. The second formula states that only two nodes have the same parent. The next two formulae ensure that each node, except the root and in the last step also its single child, have exactly one sibling and the value of the field *lft* of each node is consistent. The last formula states that all constant nodes have no children.

In addition to the properties of the expression tree Σ, we have similar conditions on the array `leaves`:

```
macro N = 20;  // number of possible expression nodes
macro L =  8;  // number of possible leaves of the expression

module EvalExpr(
  [N](nat{OpNo} * nat{N} * nat{N} * int * bool * int * int * bool) ?e,
  [L]nat{N} ?lf, int !y,event !rdy) {

  [N](nat{OpNo} * nat{N} * nat{N} * int * bool * int * int * bool) expr;
  [L]nat{N} leaf;  // list of leafs

  for(i=0..N-1) expr[i] = e[i]; // copy the inputs to local variables
  for(i=0..L-1) leaf[i] = lf[i];

  for(i=0..log(L)-1) {     // evaluation in time O(log(L))
    for(shuntLeft=0..1) { // two phase approach
      for(j=0..(L-1)/exp(2,i+1)) { // enumerates all even leafs
        let(lj = leaf[2*j])
        let(l = expr[lj])
        if(lj!=0 & l.bsy & ((shuntLeft==0) <-> l.isLeft)) {
          let(ls = l.sibling)
          let(a1 = l.a)
          let(b1 = l.b)
          let(a2 = expr[ls].a)
          let(b2 = expr[ls].b)
          let(a3 = expr[l.parent].a)
          let(b3 = expr[l.parent].b)
          let(x  = l.val) {
            next(l.bsy)= false;              // delete leaf l
            next(expr[l.parent].bsy)= false;  // delete parent of leaf l
            next(expr[ls].parent) =expr[l.parent].parent;
            next(expr[ls].sibling)=expr[l.parent].sibling;
            next(expr[ls].isLeft) =!expr[expr[l.parent].sibling].isLeft;
            next(expr[expr[l.parent].sibling].sibling) = ls;
            if (expr[l.parent].opc==ADD) {
              next(expr[ls].a) = a2 * a3;
              next(expr[ls].b) = b3 + a3 * (a1 * x + b1 + b2);
            } else {
              next(expr[ls].a) = a2 * a3 * (a1 * x + b1);
              next(expr[ls].b) = b3 + a3 * b2 * (a1 * x + b1);
      } } } }
      if(shuntLeft==1)
        for(j=0..(L/exp(2,i+1))-1)
          next(leaf[j]) = leaf[2*j+1];  // relabel the remaining leaves
    }
    pause;
  }
  let(root = expr[leaf[0]])
  y = root.val * root.a + root.b;
  emit(rdy);
}
```

Fig. 3.12: Expression Tree Evaluation - Quartz Implementation

Lemma 2. *The array* `leaves` *contains all leaves of the given expression tree Σ from left to right and the leftmost leaf is* `leaves[0]`. *Moreover, the following properties are satisfied:*

- uniqueness:
  $\forall i, j \ \mathit{with} \ (i \neq j) \wedge \Sigma[\text{leaves}[i]].bsy \wedge \Sigma[\text{leaves}[j]].bsy \ .$
  $(\text{leaves}[i] \neq \text{leaves}[j])$
- only constants:
  $\forall i \ \mathit{with} \ \Sigma[\text{leaves}[i]].bsy \ . \ \Sigma[\text{leaves}[i]].opc \equiv CST$
- completeness:
  $\forall e \in \Sigma \ \mathit{with} \ e.opc \equiv \text{CST} \wedge e.bsy . \exists i. e = \text{leaves}[i]$

This lemma states that the array `leaves` contains only pairwise different elements that are constants and no other constants exist in the expression tree.

Correctness Proof

The correctness proof with the Hoare calculus is surprisingly simple. The transformed SSTA code is shown in Figure 3.13. Due to readability, the STAs are not written as tuple assignments and instead the start and end of the STAs in the code is marked by comments. One can see that the structure is similar to the original code, only some of the actions are duplicated and/or were moved.

The only interesting issue in the correctness proof was the necessary invariant for the contained loop statement. This invariant states that the evaluation of the root node does not change during the execution. With the above mentioned assumptions, it is provable that each *shunt* operation is applied to a distinct sub-tree, hence, it is sufficient to show that for the parent of the parent of the shunt target the evaluation remains the same.

For illustration, we assume without loss of generality that the expression tree contains a sub-tree as shown in Figure 3.10. Note that swapping of e_{ps} and e_p only leads to a negation of the value in their *lft* field. The array representing this expression tree contains the following entries:

e:	o_e	p_e	s_e	v_e	l_e	a_e	b_e	t
s_e:	o_s	p_e	e	v_s	$\neg l_e$	a_s	b_s	t
p_e:	o_p	r	s_p	v_p	l_p	a_p	b_p	t
s_p:	o_{ps}	r	p_e	v_{ps}	$\neg l_p$	a_{ps}	b_{ps}	t

After the execution of the *shunt* operation on e, these entries are modified as follows (changes are marked):

e':	o_e	p_e	s_e	v_e	l_e	a_e	b_e	f
s'_e:	o_s	r	s_p	v_s	l_p	a_n	b_n	t
p'_e:	o_p	r	s_p	v_p	l_p	a_p	b_p	f
s'_p:	o_{ps}	r	s_e	v_{ps}	$\neg l_p$	a_{ps}	b_{ps}	t

```
if (0 <= log(L)-1) {
  for(i=0..log(L)-1) {
    for(shuntLeft=0..1) {
      { // *** STA 1 start
        if(i==0 & shuntLeft==0){
          for(i=0..N-1) expr[i] = e[i];
          for(i=0..L-1) leaf[i] = lf[i];
        }
        for(j=0..(L-1)/exp(2,i+1)) {
          let(lj = leaf[2*j])
          let(l = expr[lj])
          if(lj!=0 & l.bsy & ((shuntLeft==0) <-> l.isLeft)) {
           let(ls = l.sibling)
            let(a1 = l.a)
            let(b1 = l.b)
            let(a2 = expr[ls].a)
            let(b2 = expr[ls].b)
            let(a3 = expr[l.parent].a)
            let(b3 = expr[l.parent].b)
            let(x  = l.val) {
              next(l.bsy) = false;               // delete l
              next(expr[l.parent].bsy) = false;  // delete parent
              next(expr[ls].parent)  = expr[l.parent].parent;
              next(expr[ls].sibling) = expr[l.parent].sibling;
              next(expr[ls].isLeft)  = !expr[expr[l.parent].sibling].isLeft;
              next(expr[expr[l.parent].sibling].sibling) = ls;
              if (expr[l.parent].opc==ADD) {
                next(expr[ls].a) = a2 * a3;
                next(expr[ls].b) = b3 + a3 * (a1 * x + b1 + b2);
              } else {
                next(expr[ls].a) = a2 * a3 * (a1 * x + b1);
                next(expr[ls].b) = b3 + a3 * b2 * (a1 * x + b1);
        } } } }
        if(shuntLeft==1)
          for(j=0..(L/exp(2,i+1))-1)
            next(leaf[j]) = leaf[2*j+1];
      } // *** STA 1 end
      pause;
  } }
  { // *** STA 2a start
    let(root = expr[leaf[0]])
    y = root.val * root.a + root.b;
    emit(rdy);
  } // *** STA 2a end
} else
  { // *** STA 2b start
    for(i=0..N-1) expr[i] = e[i];
    for(i=0..L-1) leaf[i] = lf[i];
    let(root = expr[leaf[0]])
      y = root.val * root.a + root.b;
    emit(rdy);
  } // *** STA 2b end
```

Fig. 3.13: Module Body in STA Form

with a_n and b_n defined as follows:

- in case of an addition node ($o_p \equiv ADD$):
  $a_n = a_p * a_s$ and $b_n = b_p + a_p(a_e * v_e + b_e + b_s)$
- in case of a multiplication node ($o_p \equiv MUL$):
  $a_n = a_p * a_s(a_e * v_e + b_e)$ and $b_n = b_p + a_p * b_s(a_e * v_e + b_e)$

The difference of the evaluation for the sub-tree with root nodes r' and r, respectively, (for case $o_p \equiv ADD$ only) is computed as follows:

$$\begin{aligned}
\text{eval}(r') - \text{eval}(r) &= a'_r * v'_r + b'_r - a_r * v_r + b_r \\
&= v'_r - v_r \text{ } (a_r \text{ and } b_r \text{ are unchanged}) \\
&= (a_n * v_s + b_n + a_{ps} * v_{ps} + b_{vs}) \\
&\quad -(a_p(a_e * v_e + b_e + a_s * v_s + b_s) \\
&\quad +b_p + a_{ps} * v_{ps} + b_{vs}) \\
&= a_p * a_s * v_s + a_p(a_e * v_e + b_e + b_s) \\
&\quad +b_p - a_p(a_e * v_e + b_e + b_s) - b_p \\
&\quad -a_p * a_s * v_s \\
&= 0
\end{aligned}$$

Hence, an iteration step does not change the result of the evaluation, because the difference of the evaluation is 0 and with Lemma 1 and Lemma 2 it is provable that there are no overlapping changes.

3.2.5 Impossibility of Transforming General Quartz Programs to SSTA-Form

After having shown the usefulness of the approach this section will prove that there is no transformation that allows to rewrite arbitrary Quartz program into SSTA-form without adding additional variables. The introduction of additional variables is comparable in many ways to the elimination of goto statements in sequential programs that has been intensively discussed [BoJa66, AsMa71, BrSt72, Kosa73, Kosa76, Mill75, Elgo76] in the past. By the mentioned results, it turned out that every program with goto statements can be rewritten to an equivalent program without goto statements using only ‘structured’ statements like assignments, sequences, conditionals and loops. However, this program transformation requires the introduction of new local variables to translate the control flow to the data flow. This creates problems for the verification task as described in [KoFu06]. In particular, goto statements are simulated by assigning a corresponding local variable a value that encodes a particular location of the program. As the structure of the original program is thereby almost completely destroyed, it was a natural question whether the elimination of goto statements could still be done without using additional local variables. The question was negatively answered by different authors [AsMa71, BrSt72, Kosa73, Kosa76].

These proofs could not applied directly to synchronous programs since several ideas used there do not work for synchronous programs. In particular, the control-flow problem described in [BrSt72] cannot be created with synchronous programs. On the other hand, the problematic atomic action used in [AsMa71] that modifies the inputs during execution cannot be implemented with synchronous programs. However, it is possible to adapt the idea of the proof given in [AsMa71] in order to prove that parallel statements cannot be eliminated (without adding new variables or merging control and data flow). Thus, the program transformation to SSTA-form does not exist for arbitrary programs.

Ashcroft Manna Example Program

Ashcroft and Manna presented in [AsMa71] a program that cannot be implemented by structured statements without additional variables to store its control flow. Inspired by that program, the program shown in Figure 3.14 cannot be transformed into SSTA-form[1] without additional variables.

```
module AshcroftManna(nat{3} ?i, nat{2} !o) {
  ⎧ bool x;                     ⎫
  ⎪ o = 1;                      ⎪
  ⎪ while(!x){                  ⎪
  ⎪   while(i==0&!x){           ⎪
  ⎪     w1: pause;              ⎪
  ⎪     o = 1;                  ⎪    ⎧ do {             ⎫
  ⎪   }                         ⎪    ⎪   w5: pause;     ⎪
  ⎪   w2: pause;                ⎪    ⎪ } while(i!=2);   ⎪
  ⎨   o = 1;                    ⎬ || ⎨ w6: pause;       ⎬
  ⎪   if(!x){                   ⎪    ⎪ w7: pause;       ⎪
  ⎪     while(i==1 & !x){       ⎪    ⎩ x = true;        ⎭
  ⎪       w3: pause;            ⎪
  ⎪       o = 0;                ⎪
  ⎪     }    w4: pause;         ⎪
  ⎪     o = 0;                  ⎪
  ⎪   }                         ⎪
  ⎩ }                           ⎭
}
```

Fig. 3.14: A synchronous program that cannot be transformed to SSTA-Form.

The input of the program is a potentially infinite sequence consisting of values {0,1,2} and the output is a sequence of the same length consisting of the values 0 and 1. An extended finite state machine that can be compiled from the given program with the Quartz compiler is shown in Figure 3.15.

As can be seen, the program reads the integer variable `i` and writes to an integer output variable `o` in every macro step. Initially, it enters locations `w1`, `w5` and the assignment `o=1` is executed. After the initial point of time, the module terminates two or three steps after `i=2` holds. Moreover, if `i=0` holds, then `o=1` holds at the next but two points of time. Just as

[1] Before explaining this program, it should be emphasized that this program was the simplest one to construct with that property.

well, if `i=1` holds, then `o=0` holds at the next but two points of time, which are important properties.

The module given in Figure 3.14 satisfies some temporal logic specifications listed in Lemma 3 below, and their proof can be easily made by 'inspecting' Figure 3.15 or using a verification tool.

Lemma 3. *Module AshcroftManna satisfies the following temporal logic properties where the abbreviation* `active := **init**∨w1∨...∨w7` *is used:*

- S_1: **assert A G** (i=2 –> **A X X** (**G**¬active∨**X G**¬active))
- S_2: **assert A G** (i=0 –> **A X X** (o=1∨¬active))
- S_3: **assert A G** (i=1 –> **A X X** (o=0∨¬active))

Specification S_1 states that the program terminates two or three steps after `i=2` holds. Specification S_2 states that two steps after input i equals zero, the output equals one or the program is already terminated. The last specification is analog to the previous one.

For the following, it is important to recall that the control flow of the program remembers implicitly the last two inputs. Moreover, note that the current action of the program does not depend on the current input, but only on the previous but two inputs.

A transformation to SSTA form requires to eliminate the parallel statement, because the SSTA form implicitly forbid the use of the parallel operator (otherwise two or more STAs are executed in between two **pause** statements).

Examples for the Translation of Programs containing Parallel Statements

In particular, parallel statements are not always problematic. For example, it is no problem to replace the statement **{pause;}||{pause;}** with the behaviorally equivalent statement **pause**. Even the following program

{**while**(φ) l_1: **pause**;}||{**while**(ψ) {l_2: **pause**; l_3: **pause**;}}

that is close to the program in Figure 3.14 is reduce-able to SSTA-form without adding additional variables by removing the parallel statements as shown in Figure 3.16. This figure also shows how powerful the parallel operator is. The idea of the translation is that the outer-most loop is restarted as long as the control flow is inside both parallel threads (Line 16). As soon as one thread terminates, the remaining one is completely executed inside the loop body (Lines 7,12,17,24,29). The termination condition of each thread, which is either $\neg\varphi$ or $\neg\psi$ ensures that the other-most loop (with condition $\varphi\wedge\psi$) is left too (Lines 10,14,22,29,33).

However, *just a single pause statement behind one of the loops* makes it impossible to remove the parallel statement, which is however difficult to prove, but represents the basic idea for the proof of the following theorem.

Theorem 1 (AshcroftManna). *There exists no transformation that allows to rewrite arbitrary Quartz program into SSTA-form without adding additional variables.*

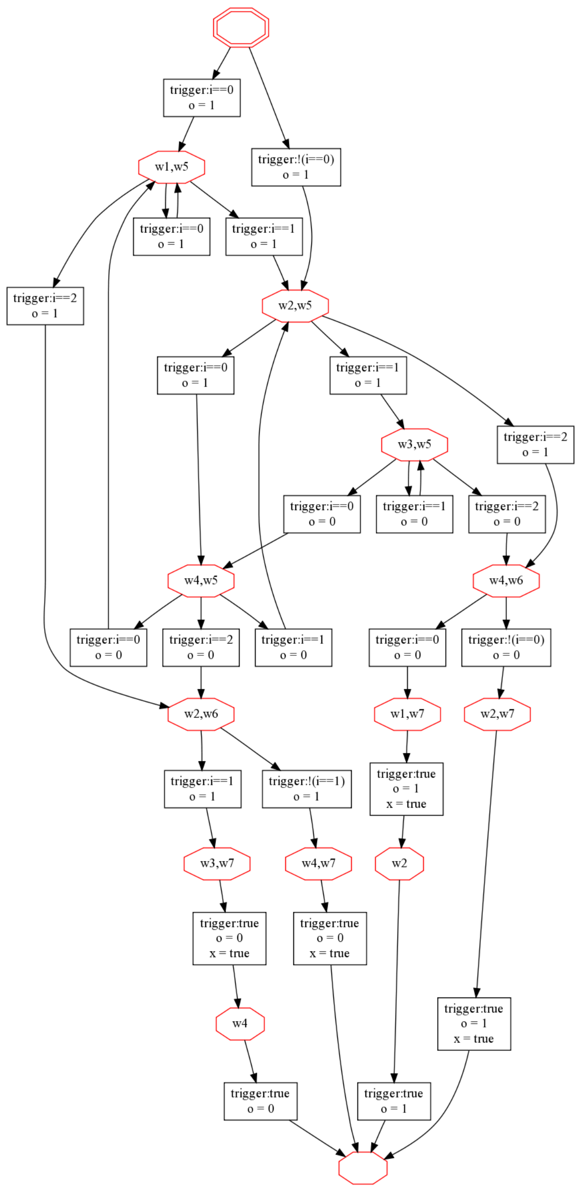


Fig. 3.15: Extended finite state machine of module AshcroftManna

Proof of Theorem 1

To derive a contradiction, assume that there exists such a transformation the generates the Quartz program `P1` in SSTA form that would be equivalent[2] to the program in Figure 3.14. Hence, `P1` must terminate two or three steps after input `i` equals two and it must generate

[2] The notion of equivalence used here is language equivalence, i.e. for all input traces the output traces of equivalent programs have to be the same.

```
do{
    if ($\varphi \wedge \psi$){ // inside both loops?
        $l_{12}$: pause;
        if ($\varphi$){ // still inside both loops?
            $l_{13}$: pause;
            if ($\psi \wedge \neg\varphi$) // leaving left
                while($\psi$){ // executing right loop
                    $l_2$: pause;
                    $l_3$: pause;
                } // $\psi$ does not hold
            else if ($\neg\psi \wedge \varphi$) // leaving right
                while ($\varphi$) // executing left loop
                    $l_1$: pause;
                // $\varphi$ does not hold
            else
                nothing; // restart execution
        } else { // leaving left & continue with right one
            $l_3$: pause;
            while($\psi$){
                $l_2$: pause;
                $l_3$: pause;
            } // $\psi$ does not hold
        }
    } else // only inside a single loop
        if ($\psi$)
            while($\psi$){
                $l_2$: pause;
                $l_3$: pause;
            } // $\psi$ does not hold
        else
            while($\varphi$)
                $l_1$: pause;
            // $\varphi$ does not hold
} while ($\varphi \wedge \psi$);
```

Fig. 3.16: SSTA Representation for the Example

the same output sequence for `o` as the original program. The program is in SSTA form that implicitly forbid the use of the parallel operator.
For the following discussion, we consider an input trace for the input i that is a potentially infinite sequence of integer values such that the k-th value in the sequence is the value of the input variable `i` in the k-th macro step. Hence, we may refer to points of time by referring to a suffix of such a sequence. We then refer to a point of time w, when at this point of time input `i` will receive the first value of the sequence w.

Since the runtime of our program depends on the input sequence, `P1` must consist of at least one loop. In particular, for every input sequence, there is a uniquely determined loop that is neither contained in another loop nor followed by another loop during the execution of the program with the considered input trace. We call that loop the *final loop for this input sequence* (similar to the original approach in [AsMa71]). Note that the parallel operator allows to run two loops in parallel, which is used in the module `AshcroftManna`, but not possible for programs in SSTA form.

To make the following considerations more concise, the following shorthand notation is introduced:

Definition 17 (Final Loop Termination). `P1:` $\alpha \hookrightarrow \beta$ *means that* if program `P1` is run with the input sequence α, it terminates the final loop executed for this input sequence at a point of time where the current input is the first value of the suffix β of α.

Then the following lemma holds:

Lemma 4. *Let* **`while`**`(`σ`)` S *be the final loop whose body statement is first entered at the point of time where the last value of sequence* $\alpha \in \mathbb{N}^*_{<2}$ *is read. For every* $n \in \mathbb{N}$*, there is a sequence* $\gamma \in \mathbb{N}^*_{<2}$ *with* $|\gamma| = n$ *such that for all sequences* $\beta \in \mathbb{N}^*_{<2}$*, we have* `P1:` $\alpha\beta\gamma 2\delta \hookrightarrow \eta\gamma 2\delta$ *where* η *is a potentially empty suffix of* $\alpha\beta$ *and* $\delta \in \mathbb{N}^*_{<3}$ *with* $2 \leq |\delta| \leq 3$ *are the inputs of the remaining macro steps until the program terminates.*

Note that for the above mentioned input sequence $\alpha\beta\gamma 2\delta$ module `AshcroftManna` terminates at the last point of time of $\alpha\beta\gamma 2\delta$ (two or three steps after reading the input $i = 2$). Since `P1` is equivalent to `AshcroftManna`, it must also terminate at this point of time.

Proof Idea

According to the above lemma, `P1` terminates its final loop even at time $\eta\gamma 2\delta$ so that the computation of `P1` must run for further $|\eta\gamma\delta| + 1$ macro steps. Since the length of γ can be made arbitrarily long, `P1` must execute a further loop which is a contradiction to the fact that the final loop has already terminated. Hence, `P1` cannot exist.

Proof of Lemma 4

Proof. The proof of the lemma is done by induction on n. In the base case, we have to prove that for all $\beta \in \mathbb{N}^*_{<2}$, we have `P1:` $\alpha\beta 2\delta \hookrightarrow \eta 2\delta$ where η is a potentially empty suffix of $\alpha\beta$. This is trivial, since the final loop must terminate at least at the final position of the input sequence, since the entire program `P1` terminates at that point of time. The last $|\delta|$ inputs are arbitrary and the previous inputs are not longer accessible and so it is impossible that the final loop ends in a step where one of them is read. This results to the fact that in the step the final loop terminates at least the suffix 2δ is left.

For the induction step, we may assume that the induction hypothesis is true for some number $n \in \mathbb{N}$, i.e. we have $\gamma \in \mathbb{N}^*_{<2}$ with $|\gamma| = n$ such that

$$(1)\ \forall \beta \in \mathbb{N}^*_{<2}. \exists \eta \in \mathbb{N}^*_{<2}.\ \texttt{P1:}\ \alpha\beta\gamma 2\delta \hookrightarrow \eta\gamma 2\delta$$

We will show that the following proof goal holds:

$$(G_1)\ \exists \gamma'.\ |\gamma'| = n + 1 \wedge \forall \beta \in \mathbb{N}^*_{<2}. \exists \eta \in \mathbb{N}^*_{<2}.\ \texttt{P1:}\ \alpha\beta\gamma' 2\delta \hookrightarrow \eta\gamma' 2\delta$$

To this end, we define a suitable witness γ' by considering the following cases:

Case 1: In this case, we assume the following

$$(2)\ \forall\beta\in\mathbb{N}^*_{<2}.\,|\beta|>0\to\exists\eta\in\mathbb{N}^*_{<2}.\ \texttt{P1:}\ \ \alpha\beta\gamma2\delta\hookrightarrow\eta\gamma2\delta\wedge|\eta|>0,$$

i.e. for all sequences $\beta\in\mathbb{N}^*_{<2}$, even more than γ, namely $\eta\gamma2\delta$ with $|\eta|>0$ is left. We can therefore define[3] $\gamma':=0\gamma$, since the final loop terminates before reaching γ. Using $\gamma':=0\gamma$ as a witness of (G_1) reduces our proof goal to $(G_2)\ \exists\eta\in\mathbb{N}^*_{<2}.\ \texttt{P1:}\ \ \alpha\beta\gamma'2\delta\hookrightarrow\eta\gamma'2\delta$ for an arbitrary sequence β. To prove this, we instantiate $\beta0$ in our case assumption (2) and obtain with a certain η with $|\eta|>0$ the fact (3) $\texttt{P1:}\ \ \alpha\beta0\gamma2\delta\hookrightarrow\eta\gamma2\delta$. Since η is nonempty and a suffix of $\alpha\beta0$, η is of the form $\eta'0$, so that we have proved that $\texttt{P1:}\ \ \alpha\beta0\gamma2\delta\hookrightarrow\eta'0\gamma2\delta$, i.e. $\texttt{P1:}\ \ \alpha\beta\gamma'2\delta\hookrightarrow\eta'\gamma'2\delta$ holds. Hence, we can use η' as a witness to prove (G_2).

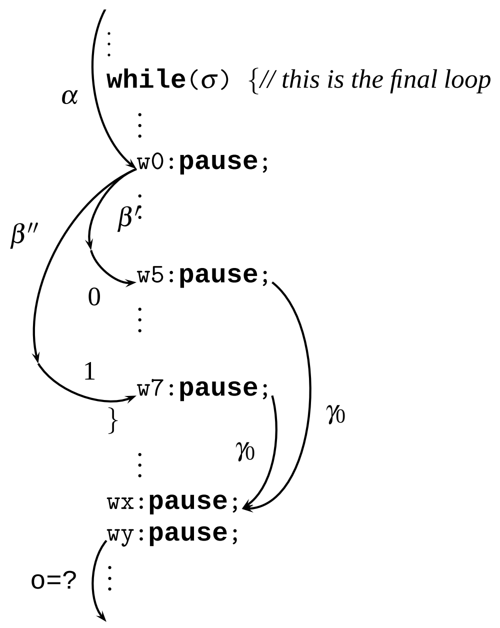


Fig. 3.17: Situation Described in the Proof of Lemma 4.

Case 2a: If (2) is not true, it follows from its negation that the following fact holds:

$$(4)\ \exists\beta\in\mathbb{N}^*_{<2}.\,|\beta|>0\wedge\texttt{P1:}\ \ \alpha\beta\gamma2\delta\hookrightarrow\gamma2\delta.$$

Hence, there are nonempty sequences β that are entirely consumed by the final loop reached by α. Assume one of these sequences β ends with 0, i.e. assume that $\beta=\beta'0$ holds. Then, we conclude (5) $\texttt{P1:}\ \ \alpha\beta'0\gamma2\delta\hookrightarrow\gamma2\delta$ by (4). In this case, we define $\gamma':=1\gamma$, so that our proof goal is reduced as follows for some arbitrary $\beta''\in\mathbb{N}^*_{<4}$:

$$(G_3)\ \exists\eta'''\in\mathbb{N}^*_{<2}.\ \texttt{P1:}\ \ \alpha\beta''1\gamma2\delta\hookrightarrow\eta'''1\gamma2\delta.$$

[3] Note that the proof can also be made with the definition $\gamma':=1\gamma$.

This means, we have to prove that the final loop does not consume the occurrence of the value 1 before γ for arbitrary $\beta'' \in \mathbb{N}^*_{<2}$. To prove (G_3), we instantiate $\beta''1$ in (1), and obtain (6) `P1:` $\alpha\beta''1\gamma2\delta \hookrightarrow \eta''\gamma2\delta$ for some η''.

We now prove that (7) $|\eta''| > 0$ holds: If η'' would be empty, it would follow by (6) that (8) `P1:` $\alpha\beta''1\gamma2\delta \hookrightarrow \gamma2\delta$ would hold. Hence, due to (5) and (8), the input sequences $\alpha\beta'0\gamma2\delta$ and $\alpha\beta''1\gamma2\delta$ will both terminate the same final loop (entered first by the last value of α) when reading the first value of $\gamma2\delta$. Clearly, the locations reached after reading the sequences $\alpha\beta'$ and $\alpha\beta''$ can be different, say `w5` and `w7`, respectively, so that the situation described is shown in Figure 3.17.

Note that the location `wx` is reached right after termination of the final loop by reading the first value γ_0 of the sequence $\gamma2\delta$. Since the program `P1` is deterministic, this location is the same for both input sequences.

Now recall that input i=0 implies that in the next but two steps the output o is set to one, and that input i=1 implies that in the next but two steps the output o is set to two. For this reason, the inputs i=0 and i=1 read in the sequences $\alpha\beta'0\gamma2$ and $\alpha\beta''1\gamma2$ right before γ that lead to the moves to locations `w5` and `w8` respectively, must lead to a one and two as output respectively, at the location `wy`. However, at location `wx` and `wy`, the reaction must only depend on the current inputs and can therefore not depend on whether this location has been reached from `w5` or `w8`. For this reason, if `P1` should be equivalent to `AshcroftManna`, we conclude that (7) $|\eta''| > 0$ holds.

Since we now know that (7) $|\eta''| > 0$ holds, and that η'' is a suffix of $\alpha\beta''1$, it must be of the form $\eta'' = \eta'''1$, so that η''' can be used as witness to prove (G_3) by the aid of (6).

Case 2b: This case is analogous to Case 2a, where we assume that none of the sequences β mentioned in (4) ends with zero, thus they are of the form $\beta'1$. We then define $\gamma' := 0\gamma$ and proceed with the proof analogously to the previous case.

■

Hence, the program `P1` and with it the assumed transformation does not exists. This means the program in Figure 3.14 is not representable by another Quartz program without adding additional variables or using the parallel operator. Hence, Theorem 1 holds. ■

This result makes the transformations presented in the next section (that were acquired before Theorem 1 was proven) less important. Nevertheless, they allow to represent a subset of all possible Quartz programs to be represented in SSTA form. Hence, for these program the presented approach is applicable. The approach presented in the following chapter emerged from the following transformations. Hence, I will describe these transformations, but will omit the complicated correctness and completeness proofs, since the approach presented in Chapter 4 obviate the need for them.

3.2.6 Transformations for a Quartz Subset to SSTA-Form

According to Theorem 1, it is not possible to eliminate parallel statements without adding additional variables or using the parallel operator. Moreover, the use of delayed assignments or abort statements in the program `P1` does not change the result since the inputs induce the next but two outputs of module `AshcroftManna`. As a result of this proof, it is also impossible to transform every program into SSTA-form in general. Nevertheless, we are able to transform all Quartz programs not containing parallel statements into SSTA-form which is shown in the following. Afterwards, we present another approach, which is able to transform some Quartz programs containing parallel statements.

Source Code Transformation Rules

In the following, we assume that the micro steps are already causally ordered by a previous causality analysis as well as the auxiliary functions $\mathsf{mkGA}\,(\Phi)$, $\mathsf{surface}\,(\Phi)$, $\mathsf{inst}\,(\Phi)$ and $\mathsf{residue}\,(\Phi)$ defined in [Schn09] are given. The function $\mathsf{mkGA}\,(\Phi)$ generates a causally ordered list of guarded actions representing the statement Φ and handles all schizophrenia problems by renaming local variables. This is essentially the compiler described in full detail in [Schn09]. A statement is split by the functions $\mathsf{surface}\,(\Phi)$ and $\mathsf{residue}\,(\Phi)$ into the surface (all micro step actions that have to be executed at starting time) and the statement that have to be executed in the next macro step. Finally, $\mathsf{inst}\,(\Phi)$ determines if the given statement consumes time or not.

Provided that ε is a Boolean value and Φ is a Quartz statement, then the following function is defined to simplify the transformation:

$$\mathsf{mkSTA}\,(\varepsilon,\Phi) := \begin{cases} \texttt{nothing}, & \varepsilon = true \\ \mathsf{mkGA}\,(\Phi), & \varepsilon = false \end{cases}$$

Unlike the function $\mathsf{mkGA}\,(\Phi)$, the function $\mathsf{mkSTA}\,(\varepsilon,\Phi)$ computes the list of guarded actions only in the case the Boolean parameter ε does not hold. Additionally, we use the function $\mathsf{splitS}\,(\Phi) := (\mathsf{surface}\,(\Phi)\,, \mathsf{inst}\,(\Phi)\,, \mathsf{residue}\,(\Phi))$ as a shortcut.

Finally, we define the function $\mathsf{CMA}\,(\varepsilon,(\Upsilon,\iota,\Phi),\Psi,\Sigma)$ to collect all micro actions into STA-blocks. Parameters of this function are the following:

- ε is a Boolean flag that indicates that all micro actions for the current macro step have already been executed (this does neither hold in the first macro step of the program nor in the first macro step of an abort statement).
- Υ contains all micro step actions of the current macro step.
- ι denotes whether the current statement consumes time (in case ι is neither true nor false, a case distinction is made by the last rule).
- Φ is the statement that is executed in the next macro step (the residual).
- Ψ defines a statement that has to be executed in every macro step of the current statement (used for abort statements).
- Σ is the statement that has to be executed after/during leaving the current statement.

The function CMA contains for each Quartz statement a case (defined by the pattern matching of Θ). It uses the function $\mathsf{splitS}\,(\Phi)$ to decompose the current macro step from the following macro steps. For $\iota = \mathsf{true}$ the micro steps contained in Υ represent the behavior of the whole macro step and will be compiled to a STA in each case of the function CMA.

Proposition 1. *Given a Quartz statement Φ that does not contain a parallel statement, the SSTA-form of the statement Φ is computed by the function*

$$\mathsf{Transform2STA}(\Phi) := \mathsf{CMA}\,(\mathsf{false}, \mathsf{splitS}\,(\Phi)\,, \textbf{\texttt{nothing}}, \textbf{\texttt{nothing}})$$

.

```
• assignment case: $\mathsf{CMA}(\varepsilon,(\varUpsilon,true,\phi),\varPsi,\varSigma) :=$
      $\mathsf{mkSTA}(\varepsilon,\varUpsilon;\ \varPsi)$; pause; $\mathsf{mkSTA}(\varepsilon,\phi;\ \varPsi;\ \varSigma)$,
      with $\phi \in \{\texttt{nothing}, [next]x = \tau\}$

• loop case: $\mathsf{CMA}(\varepsilon,(\varUpsilon,true,\varPhi_1;\texttt{while}(\sigma)\{\varPhi_2\}),\varPsi,\varSigma) :=$
      $\mathsf{CMA}(\varepsilon,(\varUpsilon,true,\varPhi_1),\varPsi,\mathsf{surface}(\texttt{while}(\sigma)\{\varPhi_2\};\ \varSigma))$;
      while($\sigma$){
          $\mathsf{CMA}(true,\mathsf{splitS}(\varPhi_2),\varPsi,\texttt{if}(\sigma)\ \mathsf{surface}(\varPhi_2)\ \texttt{else}\ \varSigma)$
      }

• sequence case: $\mathsf{CMA}(\varepsilon,(\varUpsilon,true,\varPhi_1;\ \varPhi_2),\varPsi,\varSigma) :=$
      $\mathsf{CMA}(\varepsilon,(\varUpsilon,true,\varPhi_1),\varPsi,\mathsf{surface}(\varPhi_2;\ \varSigma))$;
      $\mathsf{CMA}(true,\mathsf{splitS}(\varPhi_2),\varPsi,\varSigma)$

• abort case: $\mathsf{CMA}(\varepsilon,(\varUpsilon,true,\texttt{[weak] abort}\ \varPhi_1\ \texttt{when immediate}(\sigma)),\varPsi,\varSigma) :=$
      $\mathsf{mkSTA}(\varepsilon,\varUpsilon;\ \varPsi)$;
      [weak] abort
          $\mathsf{CMA}(true,(\varUpsilon,true,\varPhi_1),\varPsi;\texttt{if}(\sigma)\ \varSigma,\varSigma)$;
      when immediate($\sigma$)

• suspend case: $\mathsf{CMA}(\varepsilon,(\varUpsilon,true,\texttt{[weak] suspend}\ \varPhi_1\ \texttt{when immediate}(\sigma)),\varPsi,\varSigma) :=$
      $\mathsf{mkSTA}(\varepsilon,\varUpsilon;\ \varPsi)$;
      [weak] suspend
          if($\sigma$) pause; $\mathsf{mkSTA}(false,\varUpsilon;\varPsi)$;
          $\mathsf{CMA}(true,(\varUpsilon,\mathsf{inst}(\varPhi_1),\varPhi_1),\varPsi,\texttt{if}(\neg\sigma)\ \varSigma)$;
      when immediate($\sigma$)

• termination case: $\mathsf{CMA}(\varepsilon,(\varUpsilon,false,\texttt{nothing}),\varPsi,\varSigma) := \mathsf{mkSTA}(\varepsilon,\varUpsilon;\ \varPsi;\ \varSigma)$

• if-then-else case: $\mathsf{CMA}\left(\varepsilon,\left(\varUpsilon,\iota,\begin{cases}\varPhi_1, & \sigma\\ \varPhi_2, & \text{otherwise}\end{cases}\right),\varPsi,\varSigma\right) :=$
          $\mathsf{mkSTA}(\varepsilon,\varUpsilon;\ \varPsi)$;
          if($\sigma$) $\mathsf{CMA}\left(true,(\varUpsilon,[\iota]_\sigma^{true},\varPhi_1),\varPsi,\varSigma\right)$
          else $\mathsf{CMA}\left(true,(\varUpsilon,[\iota]_\sigma^{false},\varPhi_2),\varPsi,\varSigma\right)$

• if-then-else case: $\mathsf{CMA}(\varepsilon,(\varUpsilon,\iota,\varPhi),\varPsi,\varSigma) :=$
          if($\iota$) $\mathsf{CMA}(\varepsilon,(\varUpsilon,true,\varPhi),\varPsi,\varSigma)$
          else $\mathsf{CMA}(\varepsilon,(\varUpsilon,false,\varPhi),\varPsi,\varSigma)$
```

Fig. 3.18: Transformations for all Statements

Example

Figure 3.19 shows a Quartz program before (left-hand side) and after (right-hand side) applying the above defined transformation.

```
module example1 (int !a, bool ?i, ?j){
    weak abort {
        $a = 1; \ell_1$ : pause;
        while($i$)
            if($j$) {$a = 2; \ell_2$ : pause; }
            else {$a = 3; \ell_3$ : pause; }
        $a = 4$;
    }when immediate($j$);
}
```

$\underset{SSTA}{\Rightarrow}$

```
[$a = 1$]
if($\neg j$) {
    $\ell_1$ : pause;
    while($i \wedge \neg j$)
        if($j$) {
            [$a = 2$]
        else
            [$a = 3$]
        if($\neg j$) pause;
        [$a = 4$]
    }
}
```

Fig. 3.19: Example Transformation for weak abort

3.2.7 Handling Pre-emption

The function Transform2STA does not eliminate the abort and suspend statements, hence there are two ways from this point on to define a Hoare calculus for the SSTA-form: either we eliminate these statements or we introduce additional rules for them in the above transformation. The latter is quite simple, because the program is already in SSTA-form. To remove suspend statements, replace all STA-blocks Ξ after a pause statement in the scope of a weak suspend statement with condition β by:

$$\Xi;\ \textbf{while}(\beta)\{\textbf{pause};\Xi\}$$

For a strong suspend statement the block must be replaced by:

$$\textbf{while}(\beta)\{\ \textbf{pause};\}\Xi$$

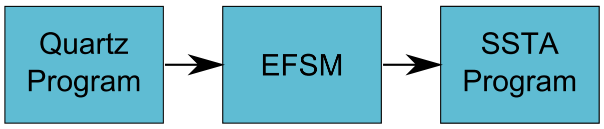
The transformation of weak abort statements with condition β is also simple: Add $\neg\beta$ as a conjunction to all loop conditions in the scope of the statement and add in front of all pause statements and all program blocks in its scope a `if`$(\neg\beta)$ that ranges to the end of a block. These if-statements jump to the end of the abort statement iff the condition β holds.

Conclusion

It was proven in Section 3.2.5 that not every program can be transformed into SSTA-form without adding additional variables. In this section a transformation was sketched that allow to transform all Quartz programs without parallel statement (there was no case definition for the parallel statement) to SSTA form. Clearly, there are programs with parallel statements that can also be transformed to SSTA-form, e.g. the programs presented in Section refsec:proofnoSSTATRans. Translating some of these programs will be explained in the following.

Transformation based on EFSM

The previous section defines a transformation of Quartz programs to SSTA-form at source-code level. This transformation is incomplete, in particular, there is no rule for the parallel statement, which is unsatisfactory since this lack prevents us from using Hoare calculus for the verification of parallel components. After some attempts to directly define such a transformation on source-code level, it was identified that the compilation to EFSMs already available in the Averest compiler is a good basis to start with: During the compilation into an EFSM, the compiler already handles problems like adding reaction-to-absence assignments, introduction of copies of local variables due to schizophrenia problems, and the resolution of causality problems. The result does not contain preemption nor parallel statements and have a simple structure, because all statements are reduced to *SGAs*. Thus, it is natural to use the EFSM output as a starting point for the program transformation into SSTA-form. This leads directly to the approach presented in this section.

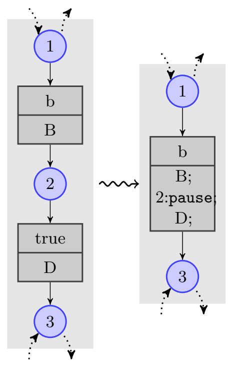


The aim is to translate this EFSM to a SSTA-form program by application of some patterns (see Figure 3.20) that reflect the control flow of the statements. Of course, due to the results in Section 3.2.5, this translation can also not be complete, but it is possible to handle some programs with parallel statements. This is very important since the parallel composition is, in practice, the key to a compositional verification, since most components of reactive systems run concurrently to each other.

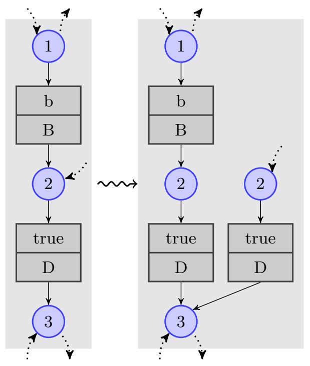
On the other hand, the translation to EFSMs unfortunately destroys some information such that the new transformation is not more powerful than the previous transformation (as demonstrated by the example of Figure 3.21). Hence, both transformations are incomparable in terms of applicability, so that a combination of both is desirable.

The transformation to SSTA-form makes use of graph rewriting using the patterns shown in Figure 3.20. During the course of graph rewriting, sub-graphs and the contained guarded actions are replaced with parts of a program in SSTA-form. For that purpose, all dependencies of the trigger condition (lower case letters) and the corresponding action list/statements (capital letters) of a transition have to be translated into statements making use of a STA. In the following, some comments on the rules of Figure 3.20 are given:

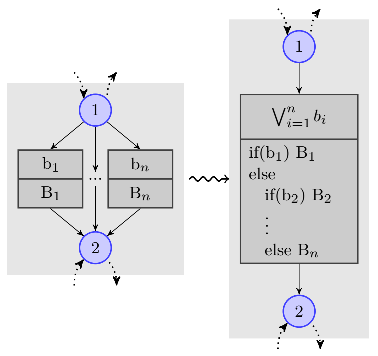
- The sequence Rule (a) reduces all sequences in the EFSM to sequences in SSTA code. Thereby the number of control flow states in the EFSM is reduced.
- Rule (b) duplicates a control flow state of the EFSM, which is sometimes necessary to make other rules match. Hence, the rule increases the number of control-flow statements.
- Rule (c) is applicable and reduces these transitions to a single one by the use of the if-then-else statement. The order of the cases is arbitrary since the guards of the transitions are mutually disjoint.
- Rule (d) is used to unroll loops which is needed, because in sequential programs without goto-statements only one loop entrance is possible. Hence, this rule transforms an EFSM that includes a cycle with more than one entrance.



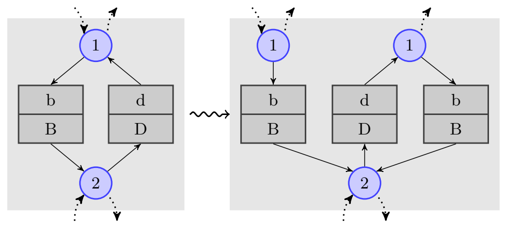
(a) *Sequence Rule*



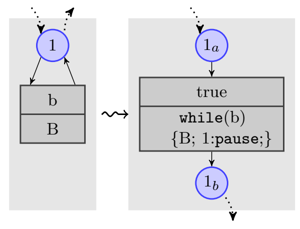
(b) *Duplication Rule*



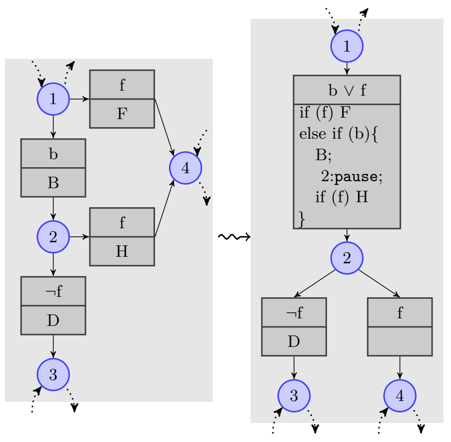
(c) *Conditional Rule*



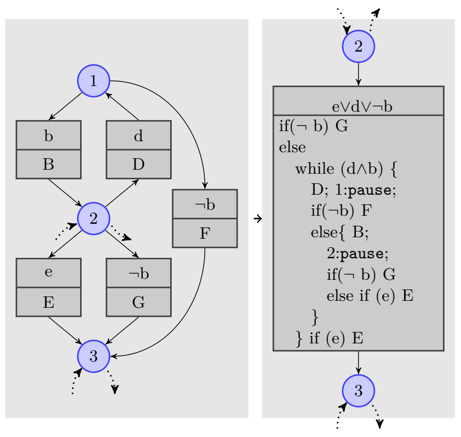
(d) *Loop Unrolling Rule*



(e) *Loop Rule*



(f) *Sequence Escape Rule*



(g) *Preemption Rule*

Fig. 3.20: Graph Rewriting Rules

```
module counterExample (bool b, c, x){
  abort
    loop
      if(b) ℓ1 : pause;
      else if(c) ℓ2 : pause;
        else ℓ3 : pause;
  } when(x);
}
```

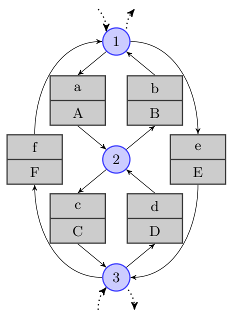


Fig. 3.21: Completeness Counter Example

- A self-loop transition is replaced by Rule (e), which introduces a while-loop statement whose body statement is the old transition's statement. The trigger condition of the new transition is true, because it is the only outgoing transition of the node and if the original trigger condition does not hold, the while loop is not entered and thus, nothing is changed.
- Rule (f) allows one to reduce step by step a sequence of an arbitrary length contained in an abort statement. Abort statements have a typical structure in the EFSM graph: All states contained in a abort statement have an outgoing transition with the same trigger condition and the same target node that does not consumes time.
- In this manner, the pre-emption Rule (g) is able to reduce a cycle in the EFSM that has several outgoing transitions that do not consume time into a SSTA code part. The condition $\neg b$ is the condition of the **abort** statement.

3.2.8 Expressiveness

The set of rules given in Figure 3.20 is not complete. For example, EFSMs that have subgraphs, whose nodes are connected with each other node, cannot be translated. An example of such an EFSM is shown in Figure 3.21, and unfortunately there are synchronous programs that generate this kind of EFSMs. Nevertheless, the rules handle most cases that occur in practice, and completeness is not possible in general due to the result of Section 3.2.5.

The advantage of the approach presented here is that the compiler automatically reduces every *serializable* statement that is not *parallel-free* to an equivalent *parallel-free*-like form.

Definition 18 (Parallel-Free Statements). *A Quartz statement that does not contain a parallel statement is* parallel-free.

Definition 19 (Serializable Statements). *A Quartz statement is* serializable *iff there exist an equivalent* parallel-free *statement.*

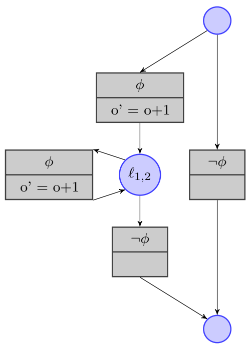
Figure 3.22 contains some examples for *serializable* statements and in Figure 3.23 the structure of the equivalent statement for the statements in Figure 3.22 is given. The only difference is the condition ϕ, which equals $a\&b$ for the first example and a for the remaining two.
It is complicated to define rules on source-code level that handle such *serializable* statements. The presented approach contains these rules implicitly, because the EFSM output of all three

```
module ma1 (bool ?a, ?b, int o) {
   {
      while(a & b){
         next(o) = o+1;
         ℓ1 : pause;
      }
   } || {
      immediate abort {
         while(b){
            next(o) = o+1;
            ℓ2 : pause;
         }
      } when(¬a);
   }
}
```

```
module ma2 (bool ?a, ?b, int o) {
   {
      while(a){
         do {
            next(o) = o+1;
            ℓ1 : pause;
         } while(a & b);
      }
   } || {
      while(a){
         next(o) = o+1;
         ℓ2 : pause;
      }
   }
}
```

```
module ma3 (bool ?a, int o) {
   immediate abort {
      {
         while(true){
            next(o) = o+1;
            ℓ1 : pause;
         }
      } || {
         while(a){
            next(o) = o+1;
            ℓ2 : pause;
         }
      }
   } when(¬a);
}
```

Fig. 3.22: Examples for *serializable* Statements



(*a*) EFSM Output Structure

```
module ma (bool ?a, ?b, int o) {
   while(φ){
      next(o) = o+1;
      ℓ1,2 : pause;
   }
}
```

(*b*) Equivalent Code Structure

Fig. 3.23: EFSM Output and Equivalent Statement

programs have the structure shown on the left-hand side of Figure 3.23, which is translatable with the loop rule. There are many more examples for *serializable* statements like these e.g. two parallel if-then-else statements, all parallel instantaneous statements and all *parallel-free* statements.

Proposition 2. *Every* parallel-free *statement is translatable to SSTA form with the source-code transformation (Proposition 1).*

Lemma 5. *Some* serializable *statements are translatable to SSTA form with the EFSM-based transformation.*

The set of rules for the EFSM-based transformation is not complete (see Figure 3.21), hence programs that contain parallel statements and are *serializable* to the program given in Figure 3.21 are *serializable* statements that the EFSM-based transformation cannot transform.

On the other hand Figure 3.22 and Figure 3.24 show some programs that are *serializable* and translatable.

3.2.9 Evaluation

```
module expl1 (bool ?i,y){
  bool a,b,x;
  {
    if(x) y = true;
    ℓ1 : pause;
    x = a;
    a = y;
  } || {
    if(i) x = true;
    if(y)
      { ℓ2 : pause; }
    else a = true;
    ℓ3 : pause;
    y = b;
  }
}
```

```
module expl2 (int[20] x,bool ?stop){
  int[20] y;
  y = 10 − x;
  abort
    do
      ℓ1 : pause;
      next(x) = x + 1;
      next(y) = y − 1;
      ℓ2 : pause;
    while(x > 10);
  when(stop);
}
```

Fig. 3.24: Examples One and Two

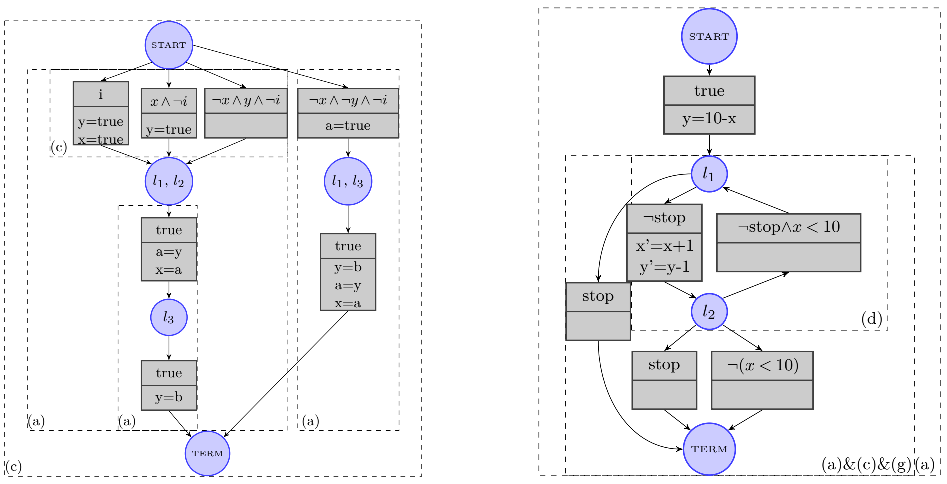


Fig. 3.25: EFSM Graphs of Examples One and Two

In this section, the application of the rules is shown by two example programs. Figure 3.24 shows two Quartz programs whose EFSMs are drawn in Figure 3.25. The first example has

```
if($\neg x \wedge \neg y \wedge \neg i$) {
  $(a).() = (true).()$;
  pause;
  $(y,a,x).() = (b,y,a).()$
} else {
  if($i$) $(x,y).() = (true,true).()$
  else if($x \wedge \neg i$) $(y).() = (true).()$;
  else if(() $\neg x \wedge y \wedge \neg i$)$().() = ().()$;
  pause;
  $(a,x).() = (y,a).()$;
  pause;
  $(y).() = (b).()$;
}
```

```
$(y).() = (10-x)()$;
pause;
if($stop$) $().() = ().()$;
else {
  $().(x,y) = ().(x+1,y-1)$;
  pause;
  if($stop$) $().() = ().()$
  else {
    if($\neg(x < 10)$) $().() = ().()$
    else while($\neg stop \wedge x < 10$){
      $().() = ().()$;
      pause;
      if($stop$) $().() = ().()$;
      else {
        $().(x,y) = ().(x+1,y-1)$;
        pause;
        if($stop$) $().() = ().()$;
        else if($\neg(x < 10)$) $().() = ().()$;
}}}}
```

Fig. 3.26: SSTA Code of Examples One and Two

an input variable `i`, and an inout variable `y`. Hence, `i` and `y` are readable and additionally `y` is also writable in the module body. Additionally the local variables `a`, `b` and `x` are used. In the parallel statement both threads read and write the variables `x` and `y` and both threads write the variable `a`. The second example contains a pre-emption statement that is able to abort the execution of the contained loop.

The result of the compilation to EFSMs is given in Figure 3.25, and the final programs in SSTA-forms are given in Figure 3.26. After adding some assertions and the translation to SSTA-form, the programs can be verified with the classical Hoare calculus and the additional rules for STAs and the pause statement.

Summary

There are many approaches to generalize the Hoare calculus to the verification of concurrent programs. However, none of them can be directly used for the verification of synchronous programs. Since synchronous programs can be translated to sequential programs, it is clear that the classic Hoare calculus can – in principle – be applied after such a compilation. However, since these translations destroy most of the syntax and since the use of the Hoare calculus is driven by the syntax, this way of using the Hoare calculus is obviously not reasonable.

For this reason, the synchronous programs are not synthesized to sequential programs to apply Hoare calculus rules, but they are rewritten into a normal form where all assignments of a macro step are combined in a single STA and no parallel operator is used. Applying this transformation manually retains most of the syntax of the synchronous programs so that the programs are kept in a readable form that is necessary for the application of interactive verification.

The transformation to SSTA form solved the problems identified during first attempts of directly define a Hoare calculus for synchronous programs. These problems are solved, because the SSTA programs behave like sequential programs. The only difference is that all assignments in a macro step are synchronously performed by the STA. All control flow statements behave as in the sequential programming model, and in particular, their control flow conditions are evaluated in the same variable environment as the assignments contained in the corresponding. Focusing on SSTA form programs, only suitable Hoare-like rules for the tuple assignments and the **pause** statements have to be defined. These rules axiomatize the synchronous model of computation.

The usefulness of the approach was shown by verifying some simple sequential examples. A more challenging example for the approach was the verification of a parallel algorithm for the evaluation of arithmetic expressions. This example emphasizes the general advantage of the Hoare calculus, namely that much larger and more abstract (in terms of data types) programs can be verified than by the use of model checking.

However, a method for interactive verification of arbitrary Quartz programs is searched. The proof of Theorem 1 showed that the approach based on the SSTA form isn't expressive enough therefore. Nevertheless, two example transformations were presented to show different ways of generating programs in SSTA form and inspire the approach presented in the next chapter. However, using these transformation to apply the defined Hoare rules require a formal correctness proof that was omitted here, since the correctness of these transformation does not have an impact on this thesis.

Chapter 4

Interactive Verification based on an Intermediate Representation

In this Chapter, the approach already published in [GeSc12a] for the interactive verification of synchronous systems is presented. The approach is based on two system representations (see Figure 4.1): the system to be verified is given as a *synchronous Quartz program* that is considered for the selection of proof rules. Then the available compiler of the Averest system is used to translate the program to the intermediate AIF representation which is essentially a set of *synchronous guarded actions*. The proof rules chosen by the user are applied to this equivalent representation. Since the obtained set of guarded actions contains not only assignments, but also *assumptions and assertions*, the guarded actions are not only used as system description, but also as *proof goals*. The user considers then the original source code and the assertions of the current proof goal to select a suitable proof rule. By the rule application, the set of guarded actions (an AIF file) is decomposed into smaller AIF files where – in analogy to the Hoare calculus – program parts are eliminated and only assumptions and assertions are left.

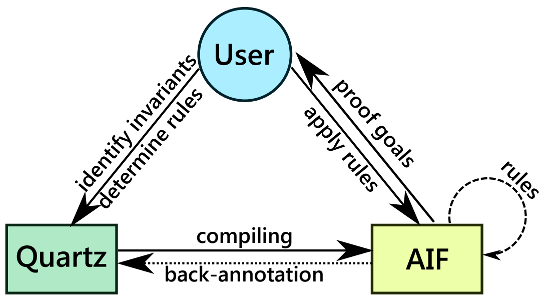


Fig. 4.1: Idea of the Approach

Due to a back-annotation via control flow locations, there is still a direct relation between the two system representations. This way, the user can still consider the more readable program code while the implementation of the proof system on top of the guarded actions allows much more flexible decomposition of the verification goals.

One can easily see that the rules are somehow inspired by rules of the Hoare calculus: *The control flow is eliminated by introduction of assumptions and assertions to achieve local provability of verification conditions.* In particular, any Hoare triple $\{\Phi\}\ S\ \{\Psi\}$ asserting that postcondition Ψ holds after termination of an non-instantaneous statement S if precondition Φ holds at starting time of S, can be expressed by adding the guarded actions $(\mathsf{enter}(S) \Rightarrow \texttt{assume}(\Phi))$ and $(\mathsf{term}(S) \Rightarrow \texttt{assert}(\Psi))$ to the set of guarded actions compiled of the statement $\mathcal{S}$. The Hoare calculus rules can then be easily implemented by the rules defined in this chapter.

In contrast to the Hoare calculus that is fully driven by the syntax of the program, the rules do not necessarily need to follow the program's syntax. Instead, the used rules focus on different decomposition techniques. For example, it is also possible to make use of techniques like *program slicing* [XQZW05, Tip95, ACHL09] by removing all guarded actions that write to variables that are not relevant for the proof. Other decomposition strategies may enumerate possible data values as suggested by McMillan e.g. in [McMi99b]. Similar to the presented approach, he focused on a combination of model-checking and theorem proving, but he had a stronger focus on model-checking. In contrast, here model-checkers are just one of several other proof procedures that can be used to check the proof tasks. McMillan also considered *refinement relations* that relate events in abstract and refined models, *circular compositional proof rules* and in particular rules for *temporal case splitting* as well as *data type reduction* and *symmetry reduction.* This work does not claim to define the final set of rules, and instead serves as basis set to be incrementally extended by adding new rules by need.

The summary of the approach is: the source code is used to select the proof rules, but the rules are applied to the corresponding set of guarded actions. The use of the two system representations (program source code and guarded actions) has many advantages:

- Difficult problems like schizophrenia problems are already solved by the compilation to guarded actions and need no longer be dealt with by the proof rules (which is a great advantage compared to the SOS rules and the idea described in Chapter 3 that have to deal with these issues).
- Due to a back-annotation via the control-flow locations, there is a strong relation between the guarded actions and the source code. Thus, the user can identify the relevant program parts for a proof goal by considering the source code.
- Using guarded actions instead of the original source code allows one more flexible decompositions of proof goals. In particular, there is no need to follow the syntax of the program.
- The compilation to guarded actions itself has been verified, hence the use of guarded actions instead of the Quartz code is therefore no correctness problem.
- Available data structures and libraries of our Averest system can be shared between compilation and verification, e.g., to optimize the code generation.
- Compared to the rich syntax at the statement level, guarded actions have a simple syntax and therefore lead to simple and only a small number of rules.
- Since guarded actions may also be considered as conditional rewrite rules, the use of term rewriting is naturally integrated in our rules. Thus, efficient theorem proving procedures are much easier to implement for guarded actions than for parallel program statements.

The outline of this chapter is as follows: in Section 4.1, we describe the approach published in [GeSc12a] and show the usefulness in Section 4.1.4. Afterwards, Section 4.2 extends the presented approach to handle temporal logic specifications given in linear temporal logic (LTL). Therefore, the notion of proof goal is extended and the rules presented in Section 4.1 are adapted.

4.1 Interactive Verification of SGAs Based on Pre- and Postconditions

We start with the definition of a proof goal and some functions used in the definition of the proof rules.

The compiled guarded actions contain besides the behavior of the program all defined assumption and assertions. Hence, the compiled AIF is directly usable as proof goal. Additionally, the part of the program currently represented by the proof goal must be bookmarked by the set of regarded control flow labels, since this information can not be computed from the AIF file. Hence, a proof goal is defined as follows:

Definition 20 (Proof Goal). *Given a Quartz program $\mathcal{S}$ and the corresponding compiled set of guarded actions $\mathcal{G}$ with control flow locations $\mathcal{L}$. The pair $(\mathcal{G},\mathcal{L})$ is a* proof goal.

For the definition of the proof rules, the following definitions are used:

Definition 21 (Control Flow Predicates). Given a proof goal $(\mathcal{G},\mathcal{L})$, we define

- $\mathsf{in}(\mathcal{G},\mathcal{L}) := \bigvee_{\ell\in\mathcal{L}} \ell$ holds if the control flow is currently at a location inside $\mathcal{L}$ and
- $\mathsf{inNxt}(\mathcal{G},\mathcal{L})$ is the disjunction of the guards γ of the control flow actions[1] of $\mathcal{G}$ $(\gamma \Rightarrow \texttt{next}(\ell) = \texttt{true})$ with $\ell \in \mathcal{L}$. Thus, $\mathsf{inNxt}(\mathcal{G},\mathcal{L})$ holds if the control flow enters next some of the locations in $\mathcal{L}$.

Based on these two definitions, there are four natural moves of the control flow:

- $\mathsf{inst}(\mathcal{G},\mathcal{L}) := \neg\mathsf{in}(\mathcal{G},\mathcal{L}) \wedge \neg\mathsf{inNxt}(\mathcal{G},\mathcal{L})$
- $\mathsf{enter}(\mathcal{G},\mathcal{L}) := \neg\mathsf{in}(\mathcal{G},\mathcal{L}) \wedge \mathsf{inNxt}(\mathcal{G},\mathcal{L})$
- $\mathsf{term}(\mathcal{G},\mathcal{L}) := \mathsf{in}(\mathcal{G},\mathcal{L}) \wedge \neg\mathsf{inNxt}(\mathcal{G},\mathcal{L})$
- $\mathsf{move}(\mathcal{G},\mathcal{L}) := \mathsf{in}(\mathcal{G},\mathcal{L}) \wedge \mathsf{inNxt}(\mathcal{G},\mathcal{L})$

$\mathsf{inst}(\mathcal{G},\mathcal{L})$ holds if the program cannot enter a location of $\mathcal{L}$ from outside, thus its execution is *instantaneous*, i.e., it does not take time. $\mathsf{enter}(\mathcal{G},\mathcal{L})$ holds if the control flow is currently outside $\mathcal{L}$, but will enter it next. $\mathsf{term}(\mathcal{G},\mathcal{L})$ holds if the control flow is currently inside $\mathcal{L}$, but will leave it next. Finally, $\mathsf{move}(\mathcal{G},\mathcal{L})$ holds if the control flow makes an internal move inside $\mathcal{L}$.

An important decomposition technique for synchronous systems is the decomposition into surface and depth which is formalized on the basis of guarded actions as follows:

[1] We note here that all control flow actions generated by the compiler are of the form $(\gamma \Rightarrow \texttt{next}(\ell) = \texttt{true})$ where ℓ is a control flow location.

Definition 22 (Surface and Depth). Given a proof goal $(\mathcal{G},\mathcal{L})$, we define its surface $\mathsf{srfc}(\mathcal{G},\mathcal{L})$ and depth $\mathsf{dpth}(\mathcal{G},\mathcal{L})$ as follows, where $\mathsf{sat}(\varphi)$ means that φ is satisfiable:

- $\mathsf{srfc}(\mathcal{G},\mathcal{L}) := \{((\gamma \Rightarrow \alpha)) \in \mathcal{G} \mid \mathsf{sat}(\neg\mathsf{in}(\mathcal{G},\mathcal{L}) \wedge \gamma)\}$
- $\mathsf{dpth}(\mathcal{G},\mathcal{L}) := \{((\gamma \Rightarrow \alpha)) \in \mathcal{G} \mid \mathsf{sat}(\mathsf{in}(\mathcal{G},\mathcal{L}) \wedge \gamma)\}$

Checking the satisfiability of program expressions is in general undecidable so that we have to approximate the above sets by means of heuristics. In case of $\mathsf{srfc}(\mathcal{G},\mathcal{L})$, this is not difficult since $\neg\mathsf{in}(\mathcal{G},\mathcal{L})$ can only be satisfied when all $\ell \in \mathcal{L}$ are made false. Thus, we can simply replace each occurrence of a label $\ell \in \mathcal{L}$ in each guard γ with false and propagate the Boolean constants. If the resulting formula is false, the action is not in the surface, otherwise we keep it (as conservative approximation) in the surface (even though the remaining guard may not be satisfiable). In a similar way, we can also approximate the depth by checking whether a guard can be satisfied when one of the locations in $\mathcal{L}$ holds: Typical guards are of the form $\ell \wedge \varphi \vee \psi$ where neither φ nor ψ contain ℓ, so that these formulas have a satisfying assignment only if there is a satisfying assignment where ℓ holds. The conditions φ,ψ that stem from control flow conditions of the program are typically satisfiable.

Example

For example, replacing all occurrences of `wa`,`wb`,`wr` in the guarded actions of ABRO by **`false`** yields:

```
control flow:
  init => next(wa)=true
  init => next(wb)=true
  false => next(wr)=true
data flow:
  false => o=true
assertions:
  false => s0:assert(a|b)
```

Thus, the surface consists of the first two control flow actions while the depth consists of all guarded actions. Splitting proof goals into their surfaces and depths is a very important decomposition that is also a key in the compilation of synchronous programs [Schn09].

4.1.1 Enumerating Control Flow States

A simple, but nevertheless effective strategy for decomposition is to enumerate the control flow states and to prove the assertions of a macro step locally in the generated states. This yields essentially an *extended finite state machine (EFSM)* that can alternatively be directly generated by the Averest compiler in that surfaces and depths are repeatedly computed. Checking a safety property can then often be done by checking the property locally in each state taking into account the assumptions and immediate assignments made in that state. In more difficult cases, a stronger property may have to be checked this way that implies the given safety property. Strengthening the proof goal is a typical step that requires user interaction.

Enumerating the reachable control flow states is often practically feasible, although this procedure is exponential in the number of control flow locations in the worst case, and thus exponential in the size of the synchronous program. In many cases, a complete enumeration is not necessary, and the defined proof rules allow one to make much shorter proofs. Many additional techniques can be used as well, and it is planned to extend the set of proof rules over time, incorporating e.g. techniques based on program slicing as discussed in [ACHL09].

Example

In case of the ABRO example, Figure 2.13 shows its corresponding EFSM with five control flow states that are defined by assignments of the control flow locations **`init`**,`wa`,`wb`, and `wr`. In each state, the data flow actions are listed that can be activated in that state, and the transitions between the states are labeled by Boolean conditions that have to hold for enabling the transition.

For example, to prove assertion `s0` of the ABRO example, the guard of the assertion has to be considerd: As can be seen in Figure 2.11, its guard obviously requires that either `a` or `b` holds. Hence, the assertion is trivially proven without enumerating all reachable states.

Another example could be that the guard of the control flow location `wr` is implied by the guard of `o`, hence the assertion **`assert`** `(o→`**`next`**`(wr))` is proven trivially by a propositional logic checker, after rewriting **`next`**`(wr)` with its definition. Checking the temporal logic specifications is more difficult and is handled later in this chapter.

4.1.2 Local Provability

In the following, we assume the existence of a procedure[2] $\mathsf{check}(\varphi)$ that will yield one of the following results for a given Boolean program expression φ:

$$\mathsf{check}(\varphi) := \begin{cases} \mathsf{1} & : \text{if } \varphi \text{ was proved to be valid} \\ \mathsf{0} & : \text{if } \varphi \text{ was proved to be invalid} \\ \bot & : \text{otherwise} \end{cases}$$

Note that the formulas φ given to the mentioned procedure $\mathsf{check}(\varphi)$ are quantifier-free Boolean program expressions. It is clear that the use of SMT solvers is recommended.

The overall task of the following proof rules is to decompose a proof goal $(\mathcal{G},\mathcal{L})$ into subgoals $(\mathcal{G}_1,\mathcal{L}_1),\ldots,(\mathcal{G}_p,\mathcal{L}_p)$ until these can be automatically proved. Note that the set of labels $\mathcal{L}_i$ of a sub-goal $(\mathcal{G}_i,\mathcal{L}_i)$ identifies the part of the program $\mathcal{S}$ that is currently considered in the subgoal. This is often a useful hint for the selection of the right proof rules.

Proving a safety property locally means checking it in each of the reachable control flow states of the EFSM. In order to make a local proof, all assumptions and assertions of a particular state have to be collected:

Definition 23. Given a proof goal $(\mathcal{G},\mathcal{L})$, we define

- $\mathsf{asm}(\mathcal{G},\mathcal{L}) := \bigwedge\{\varphi \mid ((\texttt{true} \Rightarrow \texttt{assume}(\varphi))) \in \mathcal{G}\}$ is the conjunction of all assumptions in $\mathcal{G}$ having the trivial guard **`true`**.

[2] Note that the satisfiability of program expressions is undecidable, so that the result $\bot$ of $\mathsf{check}(\varphi)$ is unavoidable.

- $\mathsf{asm}^*(\mathcal{G},\mathcal{L})$ is defined as the fixpoint of the following iteration:
  - $\mathsf{asm}^0(\mathcal{G},\mathcal{L}) := \mathsf{asm}(\mathcal{G},\mathcal{L})$
  - $\mathsf{asm}^{i+1}(\mathcal{G},\mathcal{L}) := \bigwedge\{\mathsf{cond}(\alpha) \mid \exists\gamma.\ ((\gamma \Rightarrow \alpha)) \in \mathcal{G} \wedge \mathsf{check}(\mathsf{asm}^i(\mathcal{G},\mathcal{L}) \rightarrow \gamma) = 1\}$

  where $\mathsf{cond}(\alpha)$ is defined as follows:
$$\mathsf{cond}(\alpha) := \begin{cases} \varphi & : \text{if } \alpha \equiv \texttt{\textbf{assume}}(\varphi) \\ x = \tau & : \text{if } \alpha \equiv (x = \tau) \\ \mathsf{true} & : \text{otherwise} \end{cases}$$
- $\mathsf{asr}(\mathcal{G},\mathcal{L}) := \begin{array}{l} \bigwedge\{\gamma \rightarrow \varphi \mid ((\gamma \Rightarrow \texttt{\textbf{assert}}(\varphi))) \in \mathsf{srfc}(\mathcal{G},\mathcal{L})\} \wedge \\ \bigwedge\{\texttt{next}(\gamma) \rightarrow \texttt{next}(\varphi) \mid ((\gamma \Rightarrow \texttt{\textbf{assert}}(\varphi))) \in \mathsf{dpth}(\mathcal{G},\mathcal{L})\} \end{array}$ that captures all proof obligations.

The set $\mathsf{asm}(\mathcal{G},\mathcal{L})$ is used to collect all assumptions that hold at starting time of $(\mathcal{G},\mathcal{L})$, since their guards are (syntactically) **`true`**. The formula $\mathsf{asm}^*(\mathcal{S})$ is more general in that also the transitive closure and the executed immediate assignments are collected in that condition. The task of the proof rules is to achieve that the assertions given in the program become locally provable in the following sense:

Definition 24 (Local Provability). A proof goal $(\mathcal{G},\mathcal{L})$ is locally provable if the following holds:
$$\mathsf{check}(\mathsf{asm}^*(\mathcal{G},\mathcal{L}) \rightarrow \mathsf{asr}(\mathcal{G},\mathcal{L})) = 1$$

A typical assertion given in a program will usually not be locally provable since it depends not only on the assumptions and immediate assignments of a particular state, but also on invariants that are established by the entire execution of the program. For this reason, the proof rules have the important task to add further assumptions and assertions in each decomposition step so that the finally obtained proof goals $(\mathcal{G}_i,\mathcal{L}_i)$ become locally provable.

4.1.3 Proof Rules

The decomposition rules for a synchronous system $(\mathcal{G},\mathcal{L})$ are given in Figures 4.2 and 4.4. The rules are to be read as follows: below the line, expressions like $(\mathcal{G},\mathcal{L}) \Leftarrow f(\ldots)$ denote that proof goal $(\mathcal{G},\mathcal{L})$ is decomposed into the subgoals listed above the line by applying the function f with listed arguments to $(\mathcal{G},\mathcal{L})$. Moreover, calls to $\mathsf{check}(\varphi)$ are listed above the line that have to hold for a successful rule application (otherwise, the function fails).

In addition to the already defined control flow predicates, surfaces and depths, we also assume function $\mathsf{prop}(\varphi)$ for constant propagation. Finally, $[\varphi]_x^\tau$ denotes that all occurrences of x in φ are replaced by τ.

Here are some explanations of the rules:

- Solver checks whether the current subgoal is locally provable according to Definition 24.
- Simplification eliminates all guarded actions with unsatisfiable guards.
- Branching makes a case distinction (for a conditional statement) with the given condition σ. Special cases of this rule without splitting the set of locations are the CaseDistinction rule, which allows a Boolean case distinction and the MultipleCases rules, which allows a general case distinction. Note that it is important that σ only holds at starting time, i.e., in the surface.

Solver

$$\frac{\mathsf{check}(\mathsf{asm}^*(\mathcal{G},\mathcal{L}) \rightarrow \mathsf{asr}(\mathcal{G},\mathcal{L}))}{(\mathcal{G},\mathcal{L}) \Lleftarrow \text{Solve}()}$$

Simplification

$$\frac{\begin{array}{c}(\{((\gamma \Rightarrow \alpha)) \in \mathsf{srfc}(\mathcal{G}) \mid \mathsf{check}(\mathsf{asm}^*(\mathcal{G},\mathcal{L}) \rightarrow \gamma) \neq 0\} \cup \\ \{((\gamma \Rightarrow \alpha)) \in \mathsf{dpth}(\mathcal{G}) \mid \mathsf{check}(\mathsf{asm}^*(\mathcal{G},\mathcal{L}) \rightarrow \mathtt{next}(\gamma)) \neq 0\}, \mathcal{L})\end{array}}{(\mathcal{G},\mathcal{L}) \Lleftarrow \text{Simp}()}$$

Branching

$$\frac{\begin{array}{c}\mathsf{check}(\mathsf{asm}^*(\mathcal{G} \cup \{(\mathsf{true} \Rightarrow \mathtt{assume}(\sigma))\}, \mathcal{L}) \rightarrow \mathsf{enter}\,(\mathcal{G}_1,\mathcal{L}_1)) = 1 \\ \mathsf{check}(\mathsf{asm}^*(\mathcal{G} \cup \{(\mathsf{true} \Rightarrow \mathtt{assume}(\neg\sigma))\}, \mathcal{L}) \rightarrow \mathsf{enter}\,(\mathcal{G}_2,\mathcal{L}_2)) = 1 \\ (\mathcal{G}_1 \cup \{(\mathsf{enter}\,(\mathcal{G}_1,\mathcal{L}_1) \Rightarrow \mathtt{assume}(\sigma))\}, \mathcal{L}_1) \\ (\mathcal{G}_2 \cup \{(\mathsf{enter}\,(\mathcal{G}_2,\mathcal{L}_2) \Rightarrow \mathtt{assume}(\neg\sigma))\}, \mathcal{L}_2)\end{array}}{(\mathcal{G},\mathcal{L}) \Lleftarrow \text{Branch}(\sigma,\mathcal{L}_1)}$$

with $\mathcal{L}_1 \subseteq \mathcal{L}$, $\mathcal{L}_2 = \mathcal{L} \setminus \mathcal{L}_1$ and

- $\mathcal{G}_1 := \{(\gamma \Rightarrow \alpha) \in \mathcal{G} \mid \mathsf{prop}\left(\left[[\gamma]^1_{\mathsf{l} \in \mathcal{L}_1}\right]^0_{\mathsf{l} \in \mathcal{L}_2}\right) \neq 0\}$ and
- $\mathcal{G}_2 := \{(\gamma \Rightarrow \alpha) \in \mathcal{G} \mid \mathsf{prop}\left(\left[[\gamma]^0_{\mathsf{l} \in \mathcal{L}_1}\right]^1_{\mathsf{l} \in \mathcal{L}_2}\right) \neq 0\}$

CaseDistinction

$$\frac{\begin{array}{c}(\mathcal{G} \cup \{(\mathsf{enter}\,(\mathcal{G},\mathcal{L}) \Rightarrow \mathtt{assume}(\sigma))\}, \mathcal{L}) \\ (\mathcal{G} \cup \{(\mathsf{enter}\,(\mathcal{G},\mathcal{L}) \Rightarrow \mathtt{assume}(\neg\sigma))\}, \mathcal{L})\end{array}}{(\mathcal{G},\mathcal{L}) \Lleftarrow \text{Case}(\sigma)}$$

MultipleCases

$$\frac{\begin{array}{c}\mathsf{check}(\bigvee_{\mathsf{i}=1..\mathsf{n}} \sigma_{\mathsf{i}}) = 1 \\ (\mathcal{G} \cup \{(\mathsf{enter}\,(\mathcal{G},\mathcal{L}) \Rightarrow \mathtt{assume}(\sigma_1))\}, \mathcal{L}) \\ \dots \\ (\mathcal{G} \cup \{(\mathsf{enter}\,(\mathcal{G},\mathcal{L}) \Rightarrow \mathtt{assume}(\sigma_n))\}, \mathcal{L})\end{array}}{(\mathcal{G},\mathcal{L}) \Lleftarrow \text{Cases}([\sigma_1,\dots,\sigma_n])}$$

HypothesisIntroduction

$$\frac{\begin{array}{c}\mathsf{check}(\mathsf{asm}^*(\mathcal{G} \cup \{(\mathsf{true} \Rightarrow \mathtt{assume}(\gamma))\}, \mathcal{L}) \rightarrow \varphi) = 1 \\ (\mathcal{G} \cup \{(\gamma \Rightarrow \mathtt{assume}(\varphi))\}, \mathcal{L})\end{array}}{(\mathcal{G},\mathcal{L}) \Lleftarrow \text{Hypothesize}(\gamma,\varphi)}$$

ProveAssertion

$$\frac{\begin{array}{c}\mathsf{check}(\mathsf{asm}^*(\mathcal{G} \cup \{(\mathsf{true} \Rightarrow \mathtt{assume}(\gamma))\}, \mathcal{L}) \rightarrow \varphi) = 1 \\ (\mathcal{G} \cup \{(\gamma \Rightarrow \mathtt{assume}(\varphi))\}, \mathcal{L})\end{array}}{(\{\text{assertID} : (\gamma \Rightarrow \mathtt{assert}(\varphi))\} \cup \mathcal{G}, \mathcal{L}) \Lleftarrow \text{Prove}(\text{assertID})}$$

Fig. 4.2: Proof Rules (part I)

- HypothesisIntroduction introduces a new hypothesis φ by checking the validity of the given formula and adding it afterwards to the set of guarded actions as an assumption.
- Once an assertion φ has been proven, it can be used as assumption by the ProveAssertion rule.
- The Invariant rule is similar to the invariant rule of the Hoare calculus. It is thereby assumed that the proof goal $(\mathcal{G}, \mathcal{L})$ contains as top level statement a loop with invariant γ and that the termination condition of the loop body is χ. Thus, we check that γ holds initially, and that γ also holds after each termination of the considered loop body. We may assume γ whenever the loop is iterated which is encoded by χ (end of loop body) and $\mathsf{move}(\mathcal{G}, \mathcal{L})$ (we are still in the loop or do not terminate). Notice that $\chi \wedge \mathsf{move}(\mathcal{G}, \mathcal{L}) = \mathsf{false}$ holds in the step where the loop is left.
- The SequenceSplit rule splits the current goal into two subgoals such that the considered subsystems are executed in a sequence, i.e., the second one starts as soon as the first one terminates. The given condition δ is proved after the first part of the sequence terminates, and can be used as an assumption for the second part. In Figure 4.3 an overview of this rule is given.
- The Slicing rule allows one to split the goal into two independent goals by slicing out the guarded actions that are enabled by control flow locations contained in the argument set $\mathcal{L}_1$.

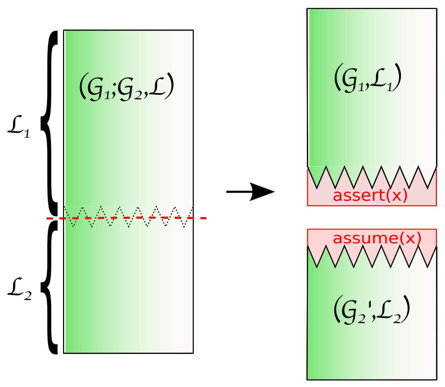

$(\mathcal{G}_1;\mathcal{G}_2,\mathcal{L})$
$\mathcal{L}_1$
$\mathcal{L}_2$
$(\mathcal{G}_1,\mathcal{L}_1)$
assert(x)
assume(x)
$(\mathcal{G}_2',\mathcal{L}_2)$

Fig. 4.3: Sequence Rule Overview

- The ConeOfInfluence rule removes all guarded actions that do not influence the assertions ($\mathsf{FV}(\varphi)$ denotes thereby the set of free variables of φ).
- The Weakening rule allows us to weaken/strengthen the pre-/postcondition.

Some of the proof rules are similar to the Hoare calculus, but they are not directly driven by the program's syntax. In many cases, it is therefore reasonable to proceed similar to Hoare calculus proofs and therefore make use of the back-annotation with regard to the original program.

Invariant

$$\frac{\begin{array}{c}\mathsf{check}(\mathsf{asm}^*(\mathcal{G},\mathcal{L}) \to \gamma) = 1 \\ (\mathcal{G} \cup \{(\mathsf{enter}\,(\mathcal{G},\mathcal{L}) \lor \chi \land \mathsf{move}\,(\mathcal{G},\mathcal{L}) \Rightarrow \mathtt{assume}(\gamma))\} \cup \{(\chi \Rightarrow \mathtt{assert}(\gamma))\}, \mathcal{L})\end{array}}{(\mathcal{G},\mathcal{L}) \Lleftarrow \mathrm{Invar}(\gamma,\chi)}$$

where χ is the termination condition of the considered top-level loop body. In case th loop is left ($\mathsf{term}\,(\mathcal{G},\mathcal{L})$ holds) the condition $\chi \land \mathsf{move}\,(\mathcal{G},\mathcal{L})$ is evaluated to false.

SequenceSplit

$$\frac{\begin{array}{c}(\mathcal{G}_1 \cup \{(\neg \mathsf{inNxt}\,(\mathcal{G}_1,\mathcal{L}_1) \Rightarrow \mathtt{assert}(\delta))\}, \mathcal{L}_1) \\ (\mathcal{G}_{2d} \cup \mathcal{G}_{2s} \cup \{(\mathsf{enter}\,(\mathcal{G}_2,\mathcal{L}_2) \Rightarrow \mathtt{assume}(\delta))\}, \mathcal{L}_2)\end{array}}{(\mathcal{G},\mathcal{L}) \Lleftarrow \mathrm{Sequence}(\mathcal{L}_1,\delta)}$$

where $\mathcal{L}_1 \subseteq \mathcal{L}$, $\mathcal{L}_2 = \mathcal{L} \setminus \mathcal{L}_1$ and

- $\mathcal{G}_1 := \{((\gamma \Rightarrow \alpha)) \in \mathcal{G} \mid \mathsf{prop}\left(\left[[\gamma]^1_{\mathsf{l}\in\mathcal{L}_1}\right]^0_{\mathsf{l}\in\mathcal{L}_2}\right) \neq 0\}$,
- $\mathcal{G}_{2s} := \{(\gamma \land \mathsf{enter}\,(\mathcal{G},\mathcal{L}_2) \land \neg\mathsf{enter}\,(\mathcal{G}_1,\mathcal{L}_1) \Rightarrow \alpha) \mid$ $((\gamma \Rightarrow \alpha)) \in \mathsf{surface}\,(\mathcal{G},\mathcal{L}) \land \mathsf{prop}\left(\left[[\gamma]^0_{\mathsf{l}\in\mathcal{L}_1}\right]^1_{\mathsf{l}\in\mathcal{L}_2}\right) \neq 0\}$ and
- $\mathcal{G}_{2d} := \{((\gamma \Rightarrow \alpha)) \in \mathsf{depth}\,(\mathcal{G},\mathcal{L}) \mid \mathsf{prop}\left(\left[[\gamma]^0_{\mathsf{l}\in\mathcal{L}_1}\right]^1_{\mathsf{l}\in\mathcal{L}_2}\right) \neq 0\}$

Slicing

$$\frac{\begin{array}{c}(\{((\gamma \Rightarrow \alpha)) \in \mathcal{G} \mid \mathsf{prop}\left(\left[[\gamma]^1_{\mathsf{l}\in\mathcal{L}_1}\right]^0_{\mathcal{L}\setminus\mathcal{L}_1}\right) \neq 0\}, \mathcal{L}_1) \\ (\{((\gamma \Rightarrow \alpha)) \in \mathcal{G} \mid \mathsf{prop}\left(\left[[\gamma]^0_{\mathsf{l}\in\mathcal{L}_1}\right]^1_{\mathsf{l}\in\mathcal{L}\setminus\mathcal{L}_1}\right) \neq 0\}, \mathcal{L} \setminus \mathcal{L}_1)\end{array}}{(\mathcal{G},\mathcal{L}) \Lleftarrow \mathrm{Slice}(\mathcal{L}_1)}$$

ConeOfInfluence

$$\frac{(\{((\gamma \Rightarrow \alpha)) \in \mathcal{G} \mid (\mathsf{FV}(\gamma) \cup \mathsf{FV}(\alpha)) \cap \mathsf{FV}(\mathsf{asr}(\mathcal{G},\mathcal{L})) \neq \emptyset\}, \mathcal{L})}{(\mathcal{G},\mathcal{L}) \Lleftarrow \mathrm{Reduce}()}$$

Weakening

$$\frac{\begin{array}{c}\mathsf{check}(\mathsf{asm}^*(\mathcal{G},\mathcal{L}) \to \varphi) = 1 \\ \mathsf{check}(\mathsf{term}\,(\mathcal{G},\mathcal{L}) \land \psi \to \mathsf{asr}(\mathcal{G},\mathcal{L})) = 1 \\ (\{(\mathsf{enter}\,(\mathcal{G},\mathcal{L}) \Rightarrow \mathtt{assume}(\varphi))\} \cup \mathcal{G} \cup \{(\mathsf{term}\,(\mathcal{G},\mathcal{L}) \Rightarrow \mathtt{assert}(\psi))\}, \mathcal{L})\end{array}}{(\mathcal{G},\mathcal{L}) \Lleftarrow \mathrm{Weaken}(\varphi,\psi)}$$

Fig. 4.4: Proof Rules (part II)

4.1.4 Evaluation

To validate this approach, the verification of some classic sequential algorithms like *computing Fibonacci numbers*, the *extended Euclidean algorithm*, and sorting algorithms like *Bubblesort* were considered.

```
macro DataWidth =  8;

module CPUInterface (
    bv{16} ?instr,               // instruction to be performed now
    nat pc,                      // program counter
    event nat !adrMem,           // address for memory access
    event bv{DataWidth} dataMem, // data for memory access
    event readMem,writeMem,      // whether data is read or written to memory
    event reqMem,ackMem,doneMem, // signals for memory transaction
    [8]bv{DataWidth} Reg         // scalar registers
    )
```

Fig. 4.5: ABACUS Interface

Additionally, the equivalence of two descriptions of a simple microprocessor called ABACUS[3] was proven. One description is a definition of its instruction set (*ScalarBehav.qrz* given in Figure 4.6) and the other one was a simple non-pipelined hardware implementation (*ScalarHW* given in Figure 4.8).

The CPUs process 16 bit instruction words on 8 bit data words. Each CPU has eight registers. The interface given in Figure 4.5 consists of a program counter `pc` and a memory access interface with address variable `adrMem`, data channel `dataMem`, flags for reading (`readMem`), writing (`writeMem`) from/to the memory and flags (`reqMem`, `ackMem`, and `doneMem`) to implement the memory protocol.

The module `CompareCPUs` in Figure 4.7 was defined to check the equivalence. It defines local variables for the CPUs' interfaces, executes both CPUs in parallel and compares the evaluation of the same `instr` in a third parallel thread. This thread demands that all contained assertions are satisfied in each macro step and therefore defines the equivalence of the processor descriptions.

One important step for the verification we identified was that the execution of the Quartz system has similarities to a global loop, even though the outermost statement is a parallel composition. The reason is that the parallel composition of two loops with the same length and execution behavior cannot be distinguished from a single loop at the level of guarded actions. Hence, the proof required an invariant for this *implicit* loop. The invariant itself is not very difficult and has many similarities with the safety condition already included in the module `CompareCPUs` (see Figure 4.10).

The proof of the equivalence was very simple, no model-checker was required to this end. Instead, only simple term rewriting was sufficient. The proof roughly made use of the invariant rule and made then case distinctions on all available machine instructions. Even though more than 40 subgoals were obtained, all of them were much smaller than the original goal since one can focus on the execution of a particular instruction. For the behavioral implementation (*ScalarBehav*) these subgoals usually only contain the actions for three assignments, as given in Figure 4.9 for the unsigned multiplication.

[3] Both descriptions are examples from the lecture 'computer systems' at the University of Kaiserslautern.

```
bv{DataWidth} overflw;       // overflw register (completing result)
bv{2*DataWidth} AluOut;      // intermediate result of ALU operations
bv{6} opc;                   // opcode of instr
nat{8} rd,rs1,rs2;           // register indices taken from instr
bv{4} cst;                   // constant operand of I-type instructions
bv{7} fnc;                   // constant operand of S-type instructions
bv{10} adr;                  // jump address of J-type instruction
bool vct;                    // whether it's a vector instruction

next(pc) = 0;
loop {
    waitInstr: pause;

    // instruction decode
    opc = OpCode(instr);
    rd  = bv2nat(DestReg(instr));
    rs1 = bv2nat(SrcLReg(instr));
    rs2 = bv2nat(SrcRReg(instr));
    cst = ConstOp(instr);
    fnc = FctCode(instr);
    adr = JumpAdr(instr);
    vct = VctFlag(instr);

    // execute current instruction
    case
        // arithmetic instructions with register operands, like:
        (opc==ADD & !vct)  do {
            AluOut = int2bv(bv2int(Reg[rs1]) + bv2int(Reg[rs2]),2*DataWidth);
            next(overflw) = UpperWord(AluOut);
            next(Reg[rd]) = LowerWord(AluOut);
            }
            ⋮
        // arithmetic instructions with constant operands
            ⋮
        // comparison instructions
            ⋮
        // logic instructions
            ⋮
        // load and store instructions
            ⋮
        // branch and jump instructions
            ⋮
    default nothing;

    // update of program counter
    if(opc!=BEZ & opc!=BNZ & opc!=JMP & opc!=J)
        next(pc) = pc+1;
}
```

Fig. 4.6: Behavioral Implementation of ABACUS

```
macro DataWidth =  8;
module CompareCPUs (bv{16} ?instr)
{
  nat pc1, pc2;
  event nat adrMem1, adrMem2;
  event bv{DataWidth} dataMem1, dataMem2;
  event readMem1,writeMem1,readMem2,writeMem2;
  event reqMem1,ackMem1,doneMem1;
  event reqMem2,ackMem2,doneMem2;
  [8]bv{DataWidth} Reg1,Reg2;

  sb: ScalarBehav(instr,pc1,adrMem1,dataMem1,
                  readMem1,writeMem1,reqMem1,
                  ackMem1,doneMem1,Reg1);
  ||
  sh: ScalarHW(instr,pc2,adrMem2,dataMem2,
               readMem2,writeMem2,reqMem2,
               ackMem2,doneMem2,Reg2);
  ||
  always{
    assert (pc1 == pc2 &
        adrMem1 == adrMem2 &
        dataMem1 == dataMem2 &
        readMem1 == readMem2 &
        writeMem1 == writeMem2 &
        reqMem1 == reqMem2 &
        ackMem1 == ackMem2 &
        doneMem1 == doneMem2);
    for (i = 0 .. 7)
        assert (Reg1[i] == Reg2[i]);
  }
}
```

Fig. 4.7: Module CompareCPUs

```
bv{DataWidth} overflw;       // overflw register (completing ALU result)
bv{6} opc;               // opcode of instr
bv{7} fnc;               // function code or 7-bit constant
nat{8} rd;               // destination register
bv{10} adr;              // jump address of J-type instruction
bv{DataWidth} opS;       // value to be stored
bv{DataWidth} opL,opR;   // ALU operands
bv{2*DataWidth} AluRes;  // ALU result
bv{DataWidth} LoadRes;   // Load result
bool CondRes;            // result of branch condition

next(pc) = 0;
loop {
    waitInstr: pause;
    Decode(instr,Reg,overflw,opc,fnc,rd,opL,opR,opS,adr);
    Execute(opc,fnc,opL,opR,AluRes,CondRes);
    MemAccess(pc,opc,fnc,opS,AluRes,LoadRes,
              adrMem,dataMem,
              readMem,writeMem,reqMem,ackMem,doneMem);
    WriteBack(pc,opc,fnc,adr,rd,AluRes,LoadRes,CondRes,Reg,overflw);
}
```

Fig. 4.8: Hardware Implementation of ABACUS

```
AluOut=nat2bv(bv2nat(Reg[rs1])*bv2nat(Reg[rs2]));
next(overflw) = UpperWord(AluOut);
next(Reg[rd]) = LowerWord(AluOut);
```

Fig. 4.9: Example: Multiplication Assignments of Behavioral Processor

The remaining subgoals are solved by term rewriting alone. Only for the memory access that involves a protocol, several case distinctions had to be done on the variables `ackMem` and `doneMem` combined with the application of the `Invariant` and `Sequence` rules.

During the verification process, several errors were identified in the implementation. First of all, signed and unsigned arithmetic operations were swapped. Then, it was identified that the instructions `BNZ` and `BEZ` used a wrong register for the comparison. After removing the bugs, the verification process was easily done and could be also used to verify the same processors with other word sizes.

Conclusions

So far the use of synchronous guarded actions as a basis for the implementation of interactive theorem provers for the verification of synchronous programs was considered. To this end, the original program is used to select suitable proof rules, but these rules were applied at the level of synchronous guarded actions that were obtained by compiling the program. The verification system is interactive, and can therefore avoid state-space explosions. Like the Hoare calculus, it can be extended in many ways, e.g. by abstracting from the size of data structures (like array sizes) as well as abstracting from the data types themselves (by considering polymorphic types). In the following section, this approach is extended to support LTL specifications.

4.2 Interactive Verification of SGAs for LTL

In this section, the approach presented in Section 4.1 and [GeSc12a] is extended to temporal logic specifications given in LTL. LTL specifications are only defined for infinite paths, hence the considered system must not terminate. Assuming that a system does not terminate, the approach presented in Section 4.1 is already applicable to handle temporal logic specifications, because an assertion $(\gamma \Rightarrow \texttt{assert}(\varphi))$ represents then the specification $\texttt{A G } (\gamma \leftarrow \varphi)$, where γ and φ are Boolean and represent a simple safety property. Therefore, this section modifies existing rules to handle other temporal logic specifications appended on a proof goal. Additionally, rules for rewriting are added. Therefore, a set of rules for already known facts about LTL formulae, rewriting rules and transformation rules will be defined.

The idea is to decompose a temporal logic specification until a propositional goal must be proven or another rule is applicable. Hence, the defined rules using the unrolling of temporal operators to distill the conditions for the current regarded macro step.

In the following, the proof rules will be motivated by describing an example and some simple rules. Afterwards, some simple decomposition rules are presented. These rules are

```
Proof "CompareCPUs" None;;

(*Instr Cases*)
let instr = GetQName("instr");;
let MkInstrSegEqu expr bL =
    BtvEqu(BtvAppend (BtvConst bL,BtvSegment (GetBtv <| expr,9,0)),
           GetBtv <| expr);;

let instr000000 i = MkInstrSegEqu i [false;false;false;false;false;false];;
let instr000001 i = MkInstrSegEqu i [false;false;false;false;false;true];;
...
let instr100111 i = MkInstrSegEqu i [true;false;false;true;true;true];;

(*List of all Instructions*)
let insrL = List.map (fun i -> i instr) [instr000000;...;instr100111];;
let InstCases = (List.map mkmap insrL)[otherwiseMap insrL];;

let fn_SYNC = Str2BoolExpr "instr{6:0} == 0b0000000";;

let rec mkRegEqu reg1 reg2 n =
    (Str2BoolExpr ("Reg1["+( n.ToString())+"] == Reg2["+(n.ToString())+"]")::(
     if (n=0)
     then [(BtvEqu(BtvNext (BtvArrAcc (reg1, GetNat<|QName2VarExpr behav_rd)),
                   BtvNext(BtvArrAcc (reg2, GetNat<|QName2VarExpr hw_rd))))]
     else mkRegEqu reg1 reg2 (n-1)) ;;

let INVAR = MkListConj (
       [
        Str2BoolExpr "next(pc1)==next(pc2)";
        Str2BoolExpr "adrMem1 == adrMem2";
        Str2BoolExpr "dataMem1 == dataMem2";
        Str2BoolExpr "readMem1 == readMem2";
        Str2BoolExpr "writeMem1 == writeMem2";
        Str2BoolExpr "reqMem1 == reqMem2";
        Str2BoolExpr "ackMem1 == ackMem2";
        Str2BoolExpr "doneMem1 == doneMem2";
       ][(BtvEqu (
            BtvNext (BtvArrAcc (QName2ArrExpr reg1, QName2NatExpr behav_rd)),
            BtvNext(BtvArrAcc (QName2ArrExpr reg2, QName2NatExpr hw_rd))))
         ]);;
MkInitialization();;
Invariant INVAR LOOPTERM();;
Sequence [] (List.map (BoolNext <| BoolVar) [l1;l4;l7]
    (List.map (fun n -> NatEqu (QName2NatExpr n, NatConst [false]))[pc1;pc2])
    (StripConj INVARSTEP));;
Solve();;

Cases InstCases;;
PrnGoal();; //40 Proof goals

ALL_AbstractArrayEquality reg1 reg2;;
ALL_Solve();; //13 Proof goals left

(*Skipped: Handling of type-casts (11 goals) and Mem access (2 goals)*)
```

Fig. 4.10: Proof for CompareCPUs

based on facts about LTL that were presented already by many authors [McQS00, BeCC98, RBHH01, KuVW01]. Here, these facts are used and integrated to show that the approach is also applicable to temproal logic specifications. Then, the running example is verified in detail.

4.2.1 Proof Goal Extension

In contrast to the approach presented in Section 4.1, the here defined rules operate on proof goals augmented by a LTL specification. In the following, we assume that synchronous systems are given as an equation system.

Definition 25 (Proof Goal). *Given a Quartz program $\mathcal{S}$, the corresponding compiled set of guarded actions $\mathcal{G}$ with control flow locations $\mathcal{L}$ and a LTL specification φ. The construct $(\mathcal{G},\mathcal{L}) \models \varphi$ is a* proof goal *and requires that* $\mathsf{term}((\mathcal{G},\mathcal{L}))$ *does not hold.*

Similar to the rules in Section 4.1 the here defined rules operate on proof goals and decompose them to a list of (sub-)goals. Obviously, the following rules are valid:

Definition 26 (Decompose). $\mathsf{Decompose}()$:

$$\frac{(\mathcal{G},\mathcal{L}) \models \gamma}{(\mathcal{G},\mathcal{L}) \models \gamma \vee \psi} \qquad \frac{(\mathcal{G},\mathcal{L}) \models \psi}{(\mathcal{G},\mathcal{L}) \models \gamma \vee \psi} \qquad \frac{(\mathcal{G},\mathcal{L}) \models \gamma \quad (\mathcal{G},\mathcal{L}) \models \psi}{(\mathcal{G},\mathcal{L}) \models \gamma \wedge \psi}$$

For a disjunction it is enough to show that a single operand holds. The operands of a conjunction are provable independently.

4.2.2 Trivial Rules

In the following some simple rules are presented to motivate the following rules.

Specifications with Always and Eventual

$$\frac{\{(\mathsf{true} \Rightarrow \mathtt{x} = \mathsf{true})\} \in \mathcal{G}}{(\mathcal{G},\mathcal{L}) \models \mathsf{G}\mathtt{x}} \qquad \frac{\{(\mathsf{true} \Rightarrow \mathtt{x} = \mathsf{true})\} \in \mathcal{G}}{(\mathcal{G},\mathcal{L}) \models \mathsf{F}\mathtt{x}}$$

$$\frac{\{(\gamma \Rightarrow \mathtt{x} = \mathsf{true})\} \in \mathcal{G} \ \wedge \ (\mathcal{G},\mathcal{L}) \models \mathsf{G}\gamma}{(\mathcal{G},\mathcal{L}) \models \mathsf{G}\mathtt{x}} \qquad \frac{\{(\gamma \Rightarrow \mathtt{x} = \mathsf{true})\} \in \mathcal{G} \ \wedge \ (\mathcal{G},\mathcal{L}) \models \mathsf{F}\gamma}{(\mathcal{G},\mathcal{L}) \models \mathsf{F}\mathtt{x}}$$

The first rule states that $\mathsf{G}\mathtt{x}$ is satisfied if $\mathcal{G}$ contains a guarded action that sets `x` and has as guard true. For the second rule it is enough if the variable is satisfied once (and the system does not terminate). Similar circumstances are given for the next two rules. The first requires that there is a guarded action that set the variable `x`, then it is enough to show that the guard holds always. The second rule requires that the guard is eventually satisfied to prove $\mathsf{F}\mathtt{x}$.

Specifications with Strong and Weak Until

The following two rules are not complete, but might easy a proof. The reason therefore is that φ may not hold always but just until ψ holds.

$$\frac{(\mathcal{G},\mathcal{L}) \models \mathsf{G}\varphi}{(\mathcal{G},\mathcal{L}) \models [\varphi \,\mathsf{U}\, \psi]} \quad \frac{(\mathcal{G},\mathcal{L}) \models \mathsf{F}\psi \wedge (\mathcal{G},\mathcal{L}) \models \mathsf{G}\varphi}{(\mathcal{G},\mathcal{L}) \models [\varphi \,\underline{\mathsf{U}}\, \psi]}$$

The first rule states that it is enough to show that the left operand of a weak until is always satisfied. The second rule requires for the strong until additionally that the second operand is eventually satisfied.

$$\frac{(\mathcal{G},\mathcal{L}) \models \psi}{(\mathcal{G},\mathcal{L}) \models [\varphi \,\mathsf{U}\, \psi]} \quad \frac{(\mathcal{G},\mathcal{L}) \models \psi}{(\mathcal{G},\mathcal{L}) \models [\varphi \,\underline{\mathsf{U}}\, \psi]}$$

These two rules require that the second operand of the until operator is directly satisfied, in that case the first operand does not matter in both cases and the behavior of both until-operators is equivalent.

4.2.3 Basic Rules

More complex rules, which are already defined in Section 4.1, are usable too:

Definition 27 (AllCases).

$$\frac{\{(\mathcal{G} \dot{\sqcup} \{(\mathsf{enter}((\mathcal{G},\mathcal{L})) \Rightarrow \texttt{assume}(\mathtt{x}\texttt{=}\tau)\},\mathcal{L}) \models \varphi \mid \tau \in \mathsf{Values}(\mathtt{x})\}}{(\mathcal{G},\mathcal{L}) \models \varphi \Leftarrow \mathrm{AllCases}(\mathtt{x})}$$

Instead of proving a single proof goal representing all possible values of a variable `x`. One may prove several proof goals each representing one of the possible values of `x`.

Definition 28 (Cases).

$$\frac{\begin{array}{c}\mathsf{check}(\bigwedge_{i=0\dots n}\sigma_i) = 1 \\ \{(\mathcal{G} \dot{\sqcup} \{(\mathsf{enter}((\mathcal{G},\mathcal{L})) \Rightarrow \texttt{assume}(\mathtt{x}\texttt{=}\sigma_i)\},\mathcal{L}) \models \varphi \mid i \in \{0\dots n\}\}\end{array}}{(\mathcal{G},\mathcal{L}) \models \varphi \Leftarrow \mathrm{Cases}([\sigma_0,\dots,\sigma_n])}$$

Instead of proving a single proof goal representing all possible values of a variable `x`. One may prove several proof goals each representing possible values of `x`.

Substitution of Variables

The given guarded actions are equations that can be used for rewriting:

$$\frac{(\texttt{true} \Rightarrow \mathtt{x} = \tau) \in \mathcal{G} \;\wedge\; (\mathcal{G},\mathcal{L}) \models [\varphi]_{\mathtt{x}}^{\tau}}{(\mathcal{G},\mathcal{L}) \models \varphi \Leftarrow \mathrm{ReWrite}(\mathtt{x})}$$

Another possibility is the introduction of an equation in the assumption to define the relation between a variable and its behavior:

$$\frac{(\texttt{true} \Rightarrow \mathtt{x} = \tau) \in \mathcal{G} \;\wedge\; (\mathcal{G},\mathcal{L}) \models \mathtt{x} = \tau \to \varphi}{(\mathcal{G},\mathcal{L}) \models \varphi \Leftarrow \mathrm{DefOf}(\mathtt{x})}$$

Unrolling of Temporal Operators

In the literature, like [Schn03], some recursion laws for the until operators are given:

$$\begin{array}{ll} [\psi\ \mathsf{U}\ \gamma] = \gamma \vee \psi \wedge \mathsf{X}\,[\psi\ \mathsf{U}\ \gamma] & [\psi\ \underline{\mathsf{U}}\ \gamma] = \gamma \vee \psi \wedge \mathsf{X}\,[\psi\ \underline{\mathsf{U}}\ \gamma] \\ [\psi\ \overleftarrow{\mathsf{U}}\ \gamma] = \gamma \vee \psi \wedge \overleftarrow{\mathsf{X}}\,[\psi\ \overleftarrow{\mathsf{U}}\ \gamma] & [\psi\ \overleftarrow{\underline{\mathsf{U}}}\ \gamma] = \gamma \vee \psi \wedge \overleftarrow{\mathsf{X}}\,[\psi\ \overleftarrow{\underline{\mathsf{U}}}\ \gamma] \end{array}$$

These rules transform the specification into a part that must be satisfied in the current state and a part containing further temporal operators. The advantage is that using these rules and the decomposition rules it is possible to decompose a proof goal to Boolean goals. Following this, we define the following rules:

Definition 29 (Unrolling Specifications). *Given a proof goal $(\mathcal{G},\mathcal{L}) \models \varphi$, the rules to unroll a specification are defined as:*

$$\frac{(\mathcal{G},\mathcal{L}) \models \gamma \vee \psi \wedge \mathsf{X}\,[\psi\ \mathsf{U}\ \gamma]}{(\mathcal{G},\mathcal{L}) \models [\psi\ \mathsf{U}\ \gamma] \Leftarrow \mathrm{NextWUntil}()}$$

$$\frac{(\mathcal{G},\mathcal{L}) \models \gamma \vee \psi \wedge \mathsf{X}\,[\psi\ \underline{\mathsf{U}}\ \gamma]}{(\mathcal{G},\mathcal{L}) \models [\psi\ \underline{\mathsf{U}}\ \gamma] \Leftarrow \mathrm{NextSUntil}()}$$

$$\frac{(\mathcal{G},\mathcal{L}) \models \gamma \vee \psi \wedge \overleftarrow{\mathsf{X}}\,[\psi\ \overleftarrow{\mathsf{U}}\ \gamma]}{(\mathcal{G},\mathcal{L}) \models [\psi\ \overleftarrow{\mathsf{U}}\ \gamma] \Leftarrow \mathrm{NextPWUntil}()}$$

$$\frac{(\mathcal{G},\mathcal{L}) \models \gamma \vee \psi \wedge \overleftarrow{\mathsf{X}}\,[\psi\ \overleftarrow{\underline{\mathsf{U}}}\ \gamma]}{(\mathcal{G},\mathcal{L}) \models [\psi\ \overleftarrow{\underline{\mathsf{U}}}\ \gamma] \Leftarrow \mathrm{NextPSUntil}()}$$

$$\frac{(\mathcal{G},\mathcal{L}) \models \neg\gamma \wedge \psi \vee \overleftarrow{\mathsf{X}}\,[\psi\ \overleftarrow{\mathsf{B}}\ \gamma]}{(\mathcal{G},\mathcal{L}) \models [\psi\ \overleftarrow{\mathsf{B}}\ \gamma] \Leftarrow \mathrm{NextPWBefore}()}$$

$$\frac{(\mathcal{G},\mathcal{L}) \models \neg\gamma \wedge \psi \vee \overleftarrow{\underline{\mathsf{X}}}\,[\psi\ \overleftarrow{\underline{\mathsf{B}}}\ \gamma]}{(\mathcal{G},\mathcal{L}) \models [\psi\ \overleftarrow{\underline{\mathsf{B}}}\ \gamma] \Leftarrow \mathrm{NextPSBefore}()}$$

Hence, unrolling $\mathcal{E} \models \mathsf{G}(\varphi)$ leads to $\mathcal{E} \models \varphi \wedge \mathsf{XG}(\varphi)$, and for $\mathcal{E} \models \mathsf{F}\varphi$ the result is $\mathcal{E} \models \varphi \vee \mathsf{XF}(\varphi)$.

Obviously, the generated proof goal for the F-operator is satisfiable either by showing that φ holds in the current step or $\mathsf{XF}\varphi$ holds in the current step, which means that the next step satisfies $\mathsf{F}\varphi$.

Correctness

The correctness of the rewriting and unrolling rules for LTL specifications are easily provable by the Quartz module given in Figure 4.11 (see `www.Averest.org/examples/Verification/TheoremProving/LTL`), where $\mathsf{A}\varphi$ is the original specification and $\mathsf{A}\psi$ the result of the rule application.

```
module CheckLTLFacts(bool a,b,c,d,...) {
    nothing;
} satisfies {
    spec: assert A (φ ↔ ψ)
}
```

Fig. 4.11: Module to Verify LTL Decomposition Rules

Inital Point of Time

Additionally, proof goals $(\mathcal{G},\mathcal{L}) \models_0 \varphi$ representing the initial point of time (InitGoals) are introduced as a special case of a proof goal that allow one to handle past operators efficiently. Note that proof goals referring to arbitrary points of time (GenGoals) are not allowed to refer to the initialization equations, and proof goals that refer to the initial point of time are moreover allowed to make use of additional proof rules like those for eliminating past temporal operators.

The following two rules concerning the initial point of time:

$$\frac{(\mathcal{G},\mathcal{L}) \models_0 \varphi \wedge (\mathcal{G},\mathcal{L}) \models \mathsf{X}\varphi}{(\mathcal{G},\mathcal{L}) \models \varphi \Leftarrow \text{TimeCases}()}$$

This rule splits a proof goal into one goal referring to the initial point of time and all other points in time.

$$\frac{(\mathcal{G},\mathcal{L}) \models \varphi}{(\mathcal{G},\mathcal{L}) \models_0 \varphi \Leftarrow \text{InitToGen}()}$$

This rule translates an InitGoal to an GenGoal.

Induction

Now it is possible to define two different induction rules, both require an always operator as outermost operator in the specification. Then the following two rules split the proof goal in an induction hypothesis and a step case.

$$\frac{(\mathcal{G},\mathcal{L}) \models \varphi \wedge (\mathcal{G},\mathcal{L}) \models \varphi \rightarrow \mathsf{X}\varphi}{(\mathcal{G},\mathcal{L}) \models \mathsf{G}\varphi \Leftarrow \text{Induction}()}$$

$$\frac{(\mathcal{G},\mathcal{L}) \models_0 \varphi \wedge (\mathcal{G},\mathcal{L}) \models \varphi \rightarrow \mathsf{X}\varphi}{(\mathcal{G},\mathcal{L}) \models \mathsf{G}\varphi \Leftarrow \text{InitInduction}()}$$

Elimination of Past Temporal Operators for InitGoals

Definition 30 (Past Operator Elimination). *Given a proof goal $(\mathcal{G},\mathcal{L}) \models \varphi$, the rules to eliminate past temporal operators are defined as:*

$$\frac{(\mathcal{G},\mathcal{L}) \models_0 \gamma \vee \psi}{(\mathcal{G},\mathcal{L}) \models_0 [\psi \overleftarrow{\mathsf{U}} \gamma] \Leftarrow \text{InitPWUntil}()}$$

$$\frac{(\mathcal{G},\mathcal{L}) \models_0 \gamma}{(\mathcal{G},\mathcal{L}) \models_0 [\psi\ \overleftarrow{\underline{\mathsf{U}}}\ \gamma] \Leftarrow \text{InitPSUntil}()}$$

$$\frac{(\mathcal{G},\mathcal{L}) \models_0 \neg\gamma}{(\mathcal{G},\mathcal{L}) \models_0 [\psi\ \overleftarrow{\mathsf{B}}\ \gamma] \Leftarrow \text{InitPWBefore}()}$$

$$\frac{(\mathcal{G},\mathcal{L}) \models_0 \neg\gamma \vee \psi}{(\mathcal{G},\mathcal{L}) \models_0 [\psi\ \overleftarrow{\underline{\mathsf{B}}}\ \gamma] \Leftarrow \text{InitPSBefore}()}$$

The first two rules remove the past-until operators by requiring that the second operand (γ) holds in the initial state. The weak variant allows additionally that only the first operand holds (ψ). In the next two rules the same circumstances are described for the past-before operators.

Example

In this section the specifications of the `ABRO` example are verified.

Proof of Specifications `s1` and `s2`

The specifications `s1` and `s2` are provable by substitution of `o` in the specification. The result of applying the substitution on `s1` and introducing the definition for the other variables leads to:

```
next(init) = False ∧
next(wb) = ¬r∧wb∧¬b ∨ r∧(wr∨wa∨wb) ∨ init ∧
next(wa) = ¬r∧wa∧¬a ∨ r∧(wr∨wa∨wb) ∨ init ∧
next(wr) = ¬r∧(wr ∨ a∧wa∧b∧wb ∨ ¬wa∧b∧wb ∨ ¬wb∧a∧wa) ⊨
¬r∧a∧wa∧b∧wb ∨ ¬r∧¬wa∧b∧wb ∨ ¬r∧¬wb∧a∧wa →
¬(
    ¬next(r)∧next(a)∧next(wa)∧next(b)∧next(wb) ∨
    ¬next(r)∧¬next(wa)∧next(b)∧next(wb) ∨
    ¬next(r)∧¬next(wb)∧next(a)∧next(wa)
)
```

A case distinction for the left hand side of the implication leads to the following three sub goals that prove the property:

```
¬r∧a∧wa∧b∧wb∧¬next(wa)∧¬next(wb)∧next(wr)⊨¬False

¬r∧¬wa∧b∧wb∧¬next(wa)∧¬next(wb)∧next(wr)⊨¬False

¬r∧¬wb∧a∧wa∧¬next(wa)∧¬next(wb)∧next(wr)⊨¬False
```

Proof of an Example Lemma

It is possible to prove the following lemma:

- Lemma0: $(\mathcal{G},\mathcal{L}) \models \mathsf{G}(\mathtt{wa} \rightarrow \overleftarrow{\mathsf{X}}[\neg\mathtt{a}\ \overleftarrow{\mathsf{U}}\ (\mathtt{r} \vee \mathbf{init})])$

The first step is the application of the InitInduction rule. The base case is trivial since wa is false at the initial point of time. In the induction step one has to show:

$$(\mathcal{G},\mathcal{L}) \models (\mathtt{wa} \rightarrow \overleftarrow{\mathsf{X}}[\neg\mathtt{a}\ \overleftarrow{\mathsf{U}}\ (\mathtt{r} \vee \mathbf{init})]) \rightarrow (\mathbf{next}(\mathtt{wa}) \rightarrow \overleftarrow{\mathsf{X}}[\neg\mathbf{next}(\mathtt{a})\ \overleftarrow{\mathsf{U}}\ (\mathbf{next}(\mathtt{r}) \vee \mathbf{next}(\mathbf{init}))]).$$

Moving $\overleftarrow{\mathsf{X}}$ inwards, where $\overleftarrow{\mathsf{X}}\mathbf{next}(\mathtt{v}) = \mathtt{v}$ for all variables $\mathtt{v} \in \mathcal{V}$ holds:

$$(\mathcal{G},\mathcal{L}) \models (\mathtt{wa} \rightarrow \overleftarrow{\mathsf{X}}[\neg\mathtt{a}\ \overleftarrow{\mathsf{U}}\ (\mathtt{r} \vee \mathbf{init})]) \rightarrow (\mathbf{next}(\mathtt{wa}) \rightarrow [\neg\mathtt{a}\ \overleftarrow{\mathsf{U}}\ (\mathtt{r} \vee \mathbf{init})])$$

Afterwards, a substitution of **next(wa)** and a simplification step are done to get:

$$(\mathcal{G},\mathcal{L}) \models (\mathtt{wa} \rightarrow \overleftarrow{\mathsf{X}}[\neg\mathtt{a}\ \overleftarrow{\mathsf{U}}\ (\mathtt{r} \vee \mathbf{init})]) \wedge (\neg\mathtt{r} \wedge \mathtt{wa} \wedge \neg\mathtt{a} \vee \mathtt{r} \wedge (\mathtt{wr} \vee \mathtt{wa} \vee \mathtt{wb}) \vee \mathbf{init}) \rightarrow [\neg\mathtt{a}\ \overleftarrow{\mathsf{U}}\ (\mathtt{r} \vee \mathbf{init})]$$

Applying NextPWUntil and discharge the generated left-hand side of the disjunction and simplify the result leads to:

$$(\mathcal{G},\mathcal{L}) \models \left(\begin{array}{c} \mathtt{wa} \rightarrow \overleftarrow{\mathsf{X}}[\neg\mathtt{a}\ \overleftarrow{\mathsf{U}}\ (\mathtt{r} \vee \mathbf{init})]) \wedge \\ (\neg\mathtt{r} \wedge \mathtt{wa} \wedge \neg\mathtt{a} \vee \mathtt{r} \wedge (\mathtt{wr} \vee \mathtt{wa} \vee \mathtt{wb}) \vee \mathbf{init}) \wedge \\ \neg\mathtt{r} \wedge \neg\mathbf{init} \end{array}\right) \rightarrow (\neg\mathtt{a} \wedge \overleftarrow{\mathsf{X}}[\neg\mathtt{a}\ \overleftarrow{\mathsf{U}}\ (\mathtt{r} \vee \mathbf{init})])$$

This is further simplified to:

$$(\mathcal{G},\mathcal{L}) \models (\mathtt{wa} \wedge \neg\mathtt{r} \wedge \neg\mathbf{init} \wedge \neg\mathtt{a} \wedge \overleftarrow{\mathsf{X}}[\neg\mathtt{a}\ \overleftarrow{\mathsf{U}}\ (\mathtt{r} \vee \mathbf{init})]) \rightarrow (\neg\mathtt{a} \wedge \overleftarrow{\mathsf{X}}[\neg\mathtt{a}\ \overleftarrow{\mathsf{U}}\ (\mathtt{r} \vee \mathbf{init})])$$

Hence, the lemma holds. ■

Proof for Specification s3

The following lemmata are proven to ease the proof of Specification s3:

- Lemma1: $(\mathcal{G},\mathcal{L}) \models \mathsf{G}(\mathtt{wa} \wedge \neg\mathtt{wb} \rightarrow \overleftarrow{\mathsf{X}}[\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}])$
- Lemma2: $(\mathcal{G},\mathcal{L}) \models \mathsf{G}(\mathtt{wb} \wedge \neg\mathtt{wa} \rightarrow \overleftarrow{\mathsf{X}}[\mathtt{a}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}])$

In the following, only the proof of Lemma1 is presented, because the proof for Lemma2 is completely symmetric. The first step is again the application of the InitInduction rule, to obtain the trivial base case (since wa does not at the initial point of time) and the following induction step:

$$(\mathcal{G},\mathcal{L}) \models (\mathtt{wa} \wedge \neg\mathtt{wb} \rightarrow \overleftarrow{\mathsf{X}}[\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}]) \rightarrow (\mathbf{next}(\mathtt{wa}) \wedge \neg\mathbf{next}(\mathtt{wb}) \rightarrow \overleftarrow{\mathsf{X}}[\mathbf{next}(\mathtt{b})\ \overleftarrow{\mathsf{B}}\ \mathbf{next}(\mathtt{r})])$$

The result after driving $\overleftarrow{\mathsf{X}}$ inwards:

$$(\mathcal{G},\mathcal{L}) \models (\mathtt{wa} \wedge \neg\mathtt{wb} \rightarrow \overleftarrow{\mathsf{X}}[\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}]) \rightarrow (\mathbf{next}(\mathtt{wa}) \wedge \neg\mathbf{next}(\mathtt{wb}) \rightarrow [\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}])$$

A rewrite step of **next(wa)** and **next(wb)** leads to:

$$(\mathcal{G},\mathcal{L}) \models \left(\begin{array}{c} \mathtt{wa} \wedge \neg\mathtt{wb} \rightarrow \overleftarrow{\mathsf{X}}[\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}] \wedge \\ \neg\mathtt{r} \wedge \mathtt{wb} \wedge \neg\mathtt{b} \vee \mathtt{r} \wedge (\mathtt{wr} \vee \mathtt{wa} \vee \mathtt{wb}) \vee \mathbf{init} \wedge \\ \neg\mathtt{r} \wedge \mathtt{wa} \wedge \neg\mathtt{a} \vee \mathtt{r} \wedge (\mathtt{wr} \vee \mathtt{wa} \vee \mathtt{wb}) \vee \mathbf{init} \end{array}\right) \rightarrow [\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}])$$

After a case distinction on **init** the case **init=false** remains:

$$(\mathcal{G},\mathcal{L}) \models \left(\begin{array}{c} \neg\mathbf{init} \wedge \mathtt{wa} \wedge \neg\mathtt{wb} \rightarrow \overleftarrow{\mathsf{X}}[\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}] \wedge \\ \neg\mathtt{r} \wedge \mathtt{wb} \wedge \neg\mathtt{b} \vee \mathtt{r} \wedge (\mathtt{wr} \vee \mathtt{wa} \vee \mathtt{wb}) \wedge \\ \neg\mathtt{r} \wedge \mathtt{wa} \wedge \neg\mathtt{a} \vee \mathtt{r} \wedge (\mathtt{wr} \vee \mathtt{wa} \vee \mathtt{wb}) \end{array}\right) \rightarrow [\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}])$$

Another case distinction on `r` allows us to eliminate the case `r=`**true**, because it contains contradicting assumptions $(\mathtt{wr} \vee \mathtt{wa} \vee \mathtt{wb})$ and $\neg(\mathtt{wr} \vee \mathtt{wa} \vee \mathtt{wb})$. The case `r=`**true** is:

$$(\mathcal{G},\mathcal{L}) \models \left(\begin{array}{c} \neg\mathbf{init} \wedge \neg\mathtt{r} \wedge \mathtt{wa} \wedge \neg\mathtt{a} \wedge \neg(\mathtt{wb} \wedge \neg\mathtt{b}) \wedge \\ \neg\mathtt{wb} \rightarrow \overleftarrow{\mathsf{X}}[\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}] \end{array}\right) \rightarrow [\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}])$$

Then, an unrolling of the past-before operator gives:

$$(\mathcal{G},\mathcal{L}) \models \left(\begin{array}{c} \neg\mathbf{init} \wedge \neg\mathtt{r} \wedge \mathtt{wa} \wedge \neg\mathtt{a} \wedge \neg(\mathtt{wb} \wedge \neg\mathtt{b}) \wedge \\ \neg\mathtt{wb} \rightarrow \overleftarrow{\mathsf{X}}[\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}] \end{array}\right) \rightarrow (\neg\mathtt{r} \wedge (\mathtt{b} \vee \overleftarrow{\mathsf{X}}[\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}])))$$

Since, $\neg\mathtt{r}$ is contained in the assumptions, the following is left to prove after a discharge step of `b`:

$$(\mathcal{G},\mathcal{L}) \models \left(\begin{array}{c} \neg\mathbf{init} \wedge \neg\mathtt{r} \wedge \mathtt{wa} \wedge \neg\mathtt{a} \wedge \neg\mathtt{b} \wedge \neg(\mathtt{wb} \wedge \neg\mathtt{b}) \wedge \\ \neg\mathtt{wb} \rightarrow \overleftarrow{\mathsf{X}}[\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}] \end{array}\right) \rightarrow \overleftarrow{\mathsf{X}}[\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}])$$

Another case distinction proves the lemma: The case $\neg\mathtt{wb}$ proves the sub goal and the case `wb` leads to the contradicting assumptions `b` and $\neg\mathtt{b}$. ■

The prove of Specification `s3` is quite simple with the help of the above lemmas:

$$(\mathcal{G},\mathcal{L}) \models \mathsf{G}(\mathtt{o} \rightarrow ([\mathtt{a}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}] \wedge [\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}]))$$

After decomposing the conjunction the following two sub goals are created:

$$(\mathcal{G},\mathcal{L}) \models \mathsf{G}(\mathtt{o} \rightarrow [\mathtt{a}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}])$$
$$(\mathcal{G},\mathcal{L}) \models \mathsf{G}(\mathtt{o} \rightarrow [\mathtt{b}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}])$$

In the following only the first sub-goal is considered further, because the second is provable analogously. Rewriting the variable `o` and several case distinctions leads to:

$$(\mathcal{G},\mathcal{L}) \models \mathsf{G}(\neg\mathbf{init} \wedge \mathtt{wa} \wedge \mathtt{wb} \wedge \neg\mathtt{r} \wedge \mathtt{a} \wedge \mathtt{b} \rightarrow [\mathtt{a}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}])$$
$$(\mathcal{G},\mathcal{L}) \models \mathsf{G}(\neg\mathbf{init} \wedge \neg\mathtt{wa} \wedge \mathtt{wb} \wedge \neg\mathtt{r} \wedge \mathtt{a} \rightarrow [\mathtt{a}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}])$$
$$(\mathcal{G},\mathcal{L}) \models \mathsf{G}(\neg\mathbf{init} \wedge \mathtt{wa} \wedge \neg\mathtt{wb} \wedge \neg\mathtt{r} \wedge \mathtt{b} \rightarrow [\mathtt{a}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}])$$

The application of the rule NextPWBefore to all sub-goals that immediately remove the conjunct $\neg\mathtt{r}$ solves the first and the last sub-goal. The second sub-goal is reduced to:

$$(\mathcal{G},\mathcal{L}) \models \mathsf{G}(\neg\mathbf{init} \wedge \neg\mathtt{wa} \wedge \mathtt{wb} \wedge \neg\mathtt{r} \wedge \mathtt{a} \wedge \neg\mathtt{b} \rightarrow \overleftarrow{\mathsf{X}}[\mathtt{a}\ \overleftarrow{\mathsf{B}}\ \mathtt{r}])$$

This allows us to use the above proven lemma (Lemma1) to solve it. In the dual case, Lemma2 must be used for the same purpose. Hence, this specification holds. ■

Conclusion

This chapter presented two approaches to interactively verify synchronous systems represented by guarded actions. The first approach is used to prove that the assertions contained in the AIF file are valid in each state of the system and the second approach is used to show that given LTL specifications are satisfied by the system. The presented results allow to implement an interactive verification tool based on a preliminary set of basic rules that could be extended by demand. Such an extension will be shown in the next chapter, where rules for module calls and preemption statements are given.

Chapter 5

Modular Verification by Decomposition of Synchronous Programs

This chapter describes rules for a special purpose – the specialization to modular verification of synchronous programs and was published in [GeSc13a, GeMS13]. In particular, techniques that allow us to reuse verification results done for modules without knowing the environment are presented. To this end, two major problems have to be solved: First, a module call may replace the formal input parameters by expressions which corresponds with a substitution of variables in the symbolic transition relation. In particular, this affects the starting point and contained pre-emption conditions of the module and can therefore dramatically affect the behavior of the module. For this reason, the temporal specifications have to be modified accordingly. A proof shows that this transformation defines a simulation preorder modulo substitution. Second, if a synchronous module is verified without its later context, outputs may not be completely determined (since the calling module may add further actions on the outputs of the called module). It is not difficult to see that the *open system* obtained by modular compilation simulates the *closed system* obtained by the linker, and therefore, all universal temporal properties are preserved. Furthermore, the behavior of synchronous modules may be temporarily suspended or finally aborted due to requests of their environment/calling module. Hence, if a temporal logic specification has already been verified for a synchronous module, then the available verification result can typically only be used if neither suspension nor abortion will take place. Therefore transformations on temporal logic specifications to lift available verification results for synchronous modules without suspension or abortion to refined temporal logic specifications are defined that take care of these pre-emption statements. In particular, the impact of a pre-emption statement of a contained module S is described. It is clear that temporal properties that hold for S may no longer be valid for the entire statement. It will be explained how to reuse available verification results for the statement S, which leads to the central question answered by this chapter: 'What is deducible for **(weak) abort** S **when**(σ) or **(weak) suspend** S **when**(σ), when S satisfies a temporal property φ?'. Therefore transformations are defined to map a temporal logic formula φ to modified temporal logic formulas $\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\varphi,\sigma)$, $\Theta_{\mathsf{ab}}^{\mathsf{st}}(\varphi,\sigma)$, $\Theta_{\mathsf{sp}}^{\mathsf{wk}}(\varphi,\sigma)$, $\Theta_{\mathsf{sp}}^{\mathsf{st}}(\varphi,\sigma)$ such that these formulas hold for **weak abort** S **when**(σ), **abort** S **when**(σ), **weak suspend** S **when**(σ), and **suspend** S **when**(σ), respectively, provided that S satisfies φ. It is clear that these formulas are equivalent to φ if σ is false, and that 'as much as possible' of φ should be retained. This way, one can establish modular verification of synchronous modules in contexts where pre-emptions are used.

The results finally determine proof rules for the verification of module calls in imperative synchronous programming languages like Esterel and Quartz.

5.1 Modular Verification of Synchronous Programs

Languages like Esterel [BeGo92] and Quartz [Schn09] allow module calls in arbitrary statements. Modules are declared with input and output parameters (variables) so that the body statement of the module is only allowed to read its input variables and to write to its output variables. If the module is later on called in a context module, the input parameters are replaced with expressions of the same type, output parameters are replaced with local or output variables of the calling module, and thus, the assignments of the called modules then become assignments of the calling module. Of course, the calling module may also make assignments to its local and output variables, so that the two behaviors are combined. Moreover, module calls can be delayed to an arbitrary point of time, and a running module can be aborted or suspended by the context of its calling module.

This semantics of module calls therefore leads to the following problems that have to be dealt with by interactive verification rules for modular verification:

- *Partially Determined Behaviors:* Since the calling module is allowed to add actions on the output variables (but not on the local variables) of the called module, it may 'complete' partially defined output traces of the called module. For this reason, a modular verification cannot state anything about the value of an output variable in case the called module does not assign a value to that output variable (we assume that there are no write conflicts with the calling module).
- *Substituted Behavior:* Since formal input/output parameters of the called module are substituted by expressions in a module call, the original behavior of the called module specializes in some way. For example, testing equality of two inputs may become constantly true or false (in case the same expression is instantiated for both inputs), so that some transitions are completely removed. The relationship between formal inputs established by this substitution can however be much more complicated. We prove in this chapter that this induces a simulation relation modulo substitution, which is enough to establish a preservation theorem for universal temporal logic similar to classic results [ClGL92, ClGL94a, LGSB95].
- *Modified Temporal Behavior:* Since module calls are allowed in every statement, a module may be started not only at the initial point of time, but also at a later point of time. Additionally, pre-emption statements may abort or temporarily suspend the execution of a module. For these reasons, the temporal logic specifications already verified for a module have to be adapted accordingly for a module call. We reduce this problem to the previous one in that we define a general context for the module that is instantiated by the module call.

The following sections will solve the above explained problems independently. Afterwards, the results of these problems are joined to reach the overall goal: interactive verification rules for modular verification of synchronous programs.

5.1.1 Partially Determined Outputs

In the following section we introduce the notion of closed and open systems including their differences, and a running example for the whole chapter.

Closed System

As explained, synchronous programs lend themselves well for formal verification, since one can easily translate synchronous programs to state transition systems so that all formal methods for transition systems can be applied. Section 2.3.6 shows how *symbolic representations of the transition systems* can be directly obtained via a compilation to synchronous guarded actions, so that formal verification methods can be directly applied. Following these steps the complete environment is fixed for these programs to get a specific behavior. In the following those systems are called closed systems, meaning that the context/environment is completely known. However, due to the state explosion problem, we have to deal with the enormous complexity of the obtained verification problems, and therefore we are interested in a modular verification. Modules obtained by modular compilation are called open systems. The environment of these systems is not known (completely) and one has to assume the most general behavior for them.

The difference of these systems is represented in a different handling of the default reaction. Since output variables of a called module are replaced in the module call by local or output variables of the called module, it is possible that the calling module may also write to the output variables of the called module. Therefore, if the called module (the regarded open system) does not assign a value to one of its output variables at a certain point of time, we cannot apply the default action as it is done in a closed system. Instead, we have to omit the default action for outputs in open systems that have to be considered for modular verification. Note, however, that the default actions have to be added for local variables of the called modules since these are not visible in the calling module.

Example

For example, consider the following Quartz program that we will use as a running example throughout the chapter:

```
module rfEdgeDet(event bool ?i,!u){
    event bool p;
    loop {
        next(p) = i;
        if(i∧¬p) u=true;
        if(¬i∧p) u=false;
        w: pause;
    }
}
```

The guarded actions that are obtained for this program are the following ones:

- **true** $\Rightarrow$ **next(init)** $=$ **false**

- $(\texttt{init} \vee \texttt{w}) \Rightarrow \texttt{next(w)} = \texttt{true}$
- $(\texttt{init} \vee \texttt{w}) \Rightarrow \texttt{next(p)} = \texttt{i}$
- $(\texttt{init} \vee \texttt{w}) \wedge \texttt{i} \wedge \neg\texttt{p} \Rightarrow \texttt{u} = \texttt{true}$
- $(\texttt{init} \vee \texttt{w}) \wedge \texttt{p} \wedge \neg\texttt{i} \Rightarrow \texttt{u} = \texttt{false}$

In addition to these explicit actions, there are also default actions that determine the values when no guard action for a variable determines its value. Note that **(init** ∨ w**)** is invariantly **true**, since **init** is **true** at the initial point of time, and `w` holds always after the initial point of time. Hence, after the initial point of time, `p` always holds the previous value of `i`, and therefore `u` is made **true** if there is a rising edge on `i`, and it is made **false** if there is a falling edge on `i`. If neither is the case, the default action takes place which resets `u` to **false** since `u` is an **event** variable.

Reconsidering the generation of a transition system introduced in Section 2.3.6 for the example module `rfEdgeDet` to generate the closed system, the following formulas are obtained where the formulas are split into the computed guarded actions and the default actions.

$$\begin{aligned}
\mathcal{I}_{\mathsf{cl}} :=\ & \begin{pmatrix} ((\texttt{init} \vee \texttt{w}) \wedge \texttt{i} \wedge \neg\texttt{p} \rightarrow \texttt{u}) \wedge \\ ((\texttt{init} \vee \texttt{w}) \wedge \texttt{p} \wedge \neg\texttt{i} \rightarrow \neg\texttt{u}) \end{pmatrix} \\
\wedge\ & \begin{pmatrix} (\texttt{true} \rightarrow \neg\texttt{p}) \wedge \\ (\texttt{true} \rightarrow \neg\texttt{w}) \wedge \\ (\texttt{true} \rightarrow \texttt{init}) \wedge \\ ((\neg\texttt{init} \wedge \neg\texttt{w}) \vee (\texttt{i} \leftrightarrow \texttt{p}) \rightarrow \neg\texttt{u}) \end{pmatrix} \\
\mathcal{R}_{\mathsf{cl}} :=\ & \begin{pmatrix} ((\texttt{init} \vee \texttt{w}) \wedge \texttt{i} \wedge \neg\texttt{p} \rightarrow \texttt{u}) \wedge \\ ((\texttt{init} \vee \texttt{w}) \wedge \texttt{p} \wedge \neg\texttt{i} \rightarrow \neg\texttt{u}) \wedge \\ (\texttt{init} \vee \texttt{w} \rightarrow \mathsf{next}(\texttt{p}) = \texttt{i}) \wedge \\ (\texttt{init} \vee \texttt{w} \rightarrow \mathsf{next}(\texttt{w})) \wedge \\ (\texttt{true} \rightarrow \mathsf{next}(\texttt{init}) = \mathsf{false}) \end{pmatrix} \\
\wedge\ & \begin{pmatrix} (\neg(\texttt{init} \vee \texttt{w}) \rightarrow \neg\mathsf{next}(\texttt{p})) \wedge \\ (\neg(\texttt{init} \vee \texttt{w}) \rightarrow \neg\mathsf{next}(\texttt{w})) \wedge \\ (\mathsf{next}((\neg\texttt{init} \wedge \neg\texttt{w}) \vee (\texttt{i} \leftrightarrow \texttt{p})) \rightarrow \neg\mathsf{next}(\texttt{u})) \end{pmatrix}
\end{aligned}$$

Figure 5.1 shows the state transition diagram of the encoded transition system for the closed system, where the reader may ignore the dashed transitions and states (they are added for the open system later). States drawn with double lines are initial states (where **init** holds), and the label of the state denotes its variable assignment. Reachable states without outgoing transitions are omitted for reasons of clarity and comprehensibility. Hence, there are two initial states and six reachable states.

Open System

Having explained the construction of symbolic transition relations for closed systems, it is almost obvious how to obtain a symbolic transition relations for open systems: All to do is to leave out the parts that model the default action for the output variables. However, the default action for local variables must be retained (like `p` and `w` in our running example).

Definition 31 (Symbolic Representation of Open Systems). *For a synchronous program with input variables $\mathcal{V}_i$, label and local variables $\mathcal{V}_l$ and output variables $\mathcal{V}_o$, we define*

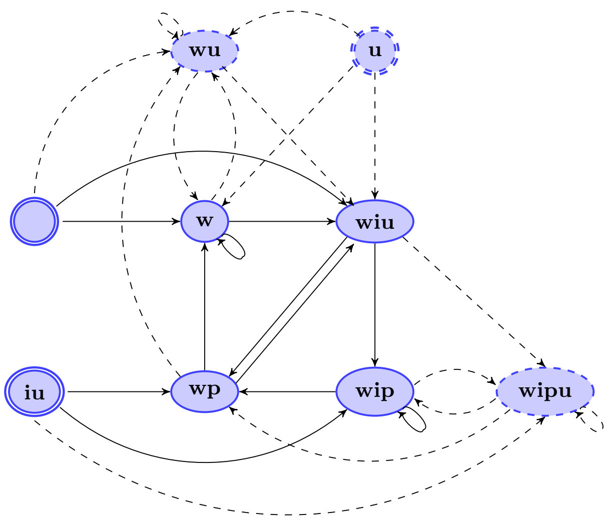


Fig. 5.1: Explicit State Transition Diagram of System `rfEdgeDet`

the initial state condition $\mathcal{I}_{\mathsf{op}}$ and the state transition relation $\mathcal{R}_{\mathsf{op}}$ of the corresponding open system as follows:

$$\begin{aligned}
\mathcal{I}_{\mathsf{op}} &:\equiv \bigwedge_{\mathtt{x}\in\mathcal{V}_l\cup\mathcal{V}_o} \mathsf{ImmActs}(\mathtt{x}) \wedge \bigwedge_{\mathtt{x}\in\mathcal{V}_l} \mathsf{InitDefActs}(\mathtt{x})\\
\mathcal{R}_{\mathsf{op}} &:\equiv \bigwedge_{\mathtt{x}\in\mathcal{V}_l\cup\mathcal{V}_o} \mathsf{ImmActs}(\mathtt{x}) \wedge \bigwedge_{\mathtt{x}\in\mathcal{V}_l\cup\mathcal{V}_o} \mathsf{DelActs}(\mathtt{x}) \wedge \bigwedge_{\mathtt{x}\in\mathcal{V}_l} \mathsf{NextDefActs}(\mathtt{x})
\end{aligned}$$

Note that the only change is that the default actions are restricted to the local variables $\mathcal{V}_l$, while in a closed system, these actions are also performed on the output variables. This is necessary, since the calling module may add further assignments to output variables which would then contradict to the default actions[1]. Therefore, output variables are not completely determined in the open system in these cases.

Hence, the open system's transition system for the `rfEdgeDet` example is given by the formulas in Figure 5.2.

The resulting transition system is shown in Figure 5.1 where this time the dashed states and transitions are added due to the deletion of the default actions.

It is clear that in general the removal of the default actions yields a transition system with more states and more transitions. Hence, we consider the following first result:

Lemma 6. *Let the two transition systems $\mathcal{T}_1 = (\mathcal{S}_1,\mathcal{I}_1,\mathcal{R}_1,\mathcal{L}_1)$ and $\mathcal{T}_2 = (\mathcal{S}_2,\mathcal{I}_2,\mathcal{R}_2,\mathcal{L}_2)$ be given where $\mathcal{S}_1 \subseteq \mathcal{S}_2$, $\mathcal{I}_1 \subseteq \mathcal{I}_2$, $\mathcal{R}_1 \subseteq \mathcal{R}_2$, and $\mathcal{L}_1(\vartheta) = \mathcal{L}_2(\vartheta)$ holds for any state $\vartheta \in \mathcal{S}_1$. Then, there exists a simulation relation $\preceq$ between $\mathcal{T}_1$ and $\mathcal{T}_2$.*

Proof. Simply define the simulation relation $\preceq$ as follows: $\vartheta_1 \preceq \vartheta_2 :\Leftrightarrow \vartheta_1 = \vartheta_2$, i.e. $\preceq$ is the identity relation. It is then clear that the properties of a simulation relation are fulfilled, i.e., we have

[1] We assume here that the system with the called module will not suffer from write conflicts.

$$\begin{array}{rl}\mathcal{I}_{\mathsf{op}} := & \left(\begin{array}{l}((\mathbf{init} \vee \mathtt{w}) \wedge \mathtt{i} \wedge \neg\mathtt{p} \to \mathtt{u}) \wedge \\ ((\mathbf{init} \vee \mathtt{w}) \wedge \mathtt{p} \wedge \neg\mathtt{i} \to \neg\mathtt{u})\end{array}\right) \\ \wedge & \left(\begin{array}{l}(\mathbf{true} \to \neg\mathtt{p}) \wedge \\ (\mathbf{true} \to \neg\mathtt{w}) \wedge \\ (\mathbf{true} \to \mathbf{init})\end{array}\right)\end{array}$$

$$\begin{array}{rl}\mathcal{R}_{\mathsf{op}} := & \left(\begin{array}{l}((\mathbf{init} \vee \mathtt{w}) \wedge \mathtt{i} \wedge \neg\mathtt{p} \to \mathtt{u}) \wedge \\ ((\mathbf{init} \vee \mathtt{w}) \wedge \mathtt{p} \wedge \neg\mathtt{i} \to \neg\mathtt{u}) \wedge \\ (\mathbf{init} \vee \mathtt{w} \to \mathsf{next}(\mathtt{p}) = \mathtt{i}) \wedge \\ (\mathbf{init} \vee \mathtt{w} \to \mathsf{next}(\mathtt{w})) \wedge \\ (\mathbf{true} \to \neg\mathsf{next}(\mathbf{init}))\end{array}\right) \\ \wedge & \left(\begin{array}{l}(\neg(\mathbf{init} \vee \mathtt{w}) \to \neg\mathsf{next}(\mathtt{p})) \wedge \\ (\neg(\mathbf{init} \vee \mathtt{w}) \to \neg\mathsf{next}(\mathtt{w}))\end{array}\right)\end{array}$$

Fig. 5.2: Transition System `rfEdgeDet` for the open system

- $\vartheta_1 \preceq \vartheta_2$ implies that for every variable $x \in \mathcal{V}$, we have $x \in \vartheta_1$ iff $x \in \vartheta_2$
- for all $\vartheta_1, \vartheta_1' \in \mathcal{S}_1$ and every $\vartheta_2 \in \mathcal{S}_2$ with $\vartheta_1 \preceq \vartheta_2$ and $(\vartheta_1, \vartheta_1') \in \mathcal{R}_1$, there is a state $\vartheta_2' \in \mathcal{S}_2$ with $\vartheta_1' \preceq \vartheta_2'$ and $(\vartheta_2, \vartheta_2') \in \mathcal{R}_2$. By definition of $\preceq$, it is clear that we choose $\vartheta_2' := \vartheta_1'$ to see this.
- for every initial state $\vartheta_1 \in \mathcal{I}_1$, there is an initial state $\vartheta_2 \in \mathcal{I}_2$ with $\vartheta_1 \preceq \vartheta_2$: clearly, we choose $\vartheta_2 := \vartheta_1$ to prove this.

By construction of $\mathcal{I}_{\mathsf{cl}}, \mathcal{R}_{\mathsf{cl}}, \mathcal{I}_{\mathsf{op}}$ and $\mathcal{R}_{\mathsf{op}}$, it is clear that we have $\mathcal{I}_{\mathsf{cl}} \subseteq \mathcal{I}_{\mathsf{op}}$ and $\mathcal{R}_{\mathsf{cl}} \subseteq \mathcal{R}_{\mathsf{op}}$, so that we can apply the above lemma. Hence, the open system simulates the closed system, and therefore, we have the following theorem (see [GrLo91] for a definition of ACTL^*):

Theorem 2. *Let $\mathcal{T}_{\mathsf{op}}$ and $\mathcal{T}_{\mathsf{cl}}$ be the open and the closed transition system obtained for a synchronous program $\mathcal{P}$. For every universal temporal logic formula $\varphi \in \mathsf{ACTL}^*$ with $\mathcal{T}_{\mathsf{op}} \models \varphi$, we then also have $\mathcal{T}_{\mathsf{cl}} \models \varphi$, i.e., we have the following proof rules (where ϑ is any state in $\mathcal{S}_{\mathsf{cl}}$):*

$$\frac{\mathcal{T}_{\mathsf{op}} \models \varphi}{\mathcal{T}_{\mathsf{cl}} \models \varphi} \qquad\qquad \frac{\mathcal{T}_{\mathsf{op}}, \vartheta \models \varphi}{\mathcal{T}_{\mathsf{cl}}, \vartheta \models \varphi}$$

Illustrating Example

Considering the running example `rfEdgeDet` and its corresponding closed system $\mathcal{T}_{\mathsf{cl}}$ and open system $\mathcal{T}_{\mathsf{op}}$ to illustrate the use of the above proof rules. It is not difficult to verify that the open system $\mathcal{T}_{\mathsf{op}}$ satisfies the following specifications:

- $\Phi_0 := \mathbf{A}\ \mathbf{G}\ (\neg\mathtt{i} \wedge \mathbf{X}\ (\mathtt{i} \to \mathtt{u}))$
- $\Phi_1 := \mathbf{A}\ \mathbf{G}\ (\mathtt{i} \wedge \mathbf{X}\ (\neg\mathtt{i} \to \neg\mathtt{u}))$

Φ_0 states that `u` must be true whenever there is a rising edge on `i`, and Φ_1 states that `u` must be false whenever there is a falling edge on `i`. In this case, the specifications are very close to the implementation, which is in general, of course, not necessary. By Theorem 2, we know that these specifications are also valid for each system obtained from $\mathcal{T}_{\mathsf{op}}$ by adding new guarded actions that restrict the behavior, such as $\mathcal{T}_{\mathsf{cl}}$.

The reverse implication is in general not true: There are temporal logic formulas that hold in $\mathcal{T}_{\mathsf{cl}}$, but not in $\mathcal{T}_{\mathsf{op}}$. Since the closed system sets u whenever a rising edge on i is detected, and to **false** either by the default action or by the assignment triggered when a falling edge on i is detected, and the fact that in between two rising edges, a falling edge must occur, it is possible to verify that the following specifications hold for $\mathcal{T}_{\mathsf{cl}}$:

- $\Phi_2 := \mathbf{A}\ \mathbf{G}\ [\neg \mathtt{u}\ \mathtt{U}\ (\neg \mathtt{i} \wedge \mathbf{X}\ \mathtt{i})]$
- $\Phi_3 := \mathbf{A}\ \mathbf{G}\ (\mathtt{u} \rightarrow \mathbf{X}\ [\neg \mathtt{u}\ \mathtt{U}\ (\mathtt{i} \wedge \mathbf{X} \neg \mathtt{i})])$

Φ_2 states that u is false until there is a rising edge on i, and Φ_3 states that whenever u holds, it will remain false until there is a falling edge on i. Note that this holds in the closed system due to the default action on u that takes place whenever there is neither a rising nor a falling edge on i. The default action is however not present in the open system $\mathcal{T}_{\mathsf{op}}$, and therefore arbitrary values are allowed there for u in these cases. Hence, neither Φ_2 nor Φ_3 holds for the open system $\mathcal{T}_{\mathsf{op}}$.

This can be seen in the transition system given in Figure 5.1: a counterexample for Φ_2 is the path $\pi := \{\mathbf{init},\mathtt{u}\},\{\mathtt{w},\mathtt{u}\}^{\omega}$, where s^{ω} expresses the infinite repetition of s to obtain an infinite path. Using $\{init\},\{\mathtt{w},\mathtt{u}\}^{\omega}$, we are also able to find a counterexample starting in an initial state of the closed system (but we need to use at least one transition of $\mathcal{T}_{\mathsf{op}}$, since Φ_2 does hold in $\mathcal{T}_{\mathsf{cl}}$). The path π is also a counterexample for the specification Φ_3.

5.1.2 Simulation Modulo Substitution

Since module calls are allowed in arbitrary statements, it is possible that a module call is started after the initial point of time or inside a pre-emption statement that can abort or suspend the called module. In [Schn09, BrSc09], a general interface for modular compilation has been introduced to handle these cases. In essence, the modular compilation of a given module M with interface I is based on the consideration of the Quartz program given in Figure 5.3 that embeds the module in a compilation context. As can be seen, this compilation context consists of a nesting of abortion and suspension statements, and additionally, a wait statement is placed in front of this nesting. Hence, the entire statement waits until st holds, then calls the module, and if during the execution of the module one of the pre-emption conditions abrt, wabrt, susp, or wsusp holds, then the corresponding pre-emption takes place. The pre-emption conditions abrt, wabrt, susp, and wsusp as well as the start condition st used in this interface are new input signals. In a later module call, the calling module provides conditions for these variables that are used by the linker to replace the preliminary pre-emption conditions.

Clearly, the synchronous guarded actions obtained by the compiler will therefore refer to abrt, wabrt, susp, wsusp and st in addition to the variables declared in the module. For the running example rfEdgeDet, the obtained guarded actions are listed in Figure 5.4. Besides the already mentioned variables (st,abrt,wabrt,susp, and wsusp) also some new label variables are introduced, which represent that either the module is not yet started (nst) or a (strong) suspension takes place during the module's start (lss and lws).
Furthermore, a module call defines a substitution of the input variables (including the above variables for the compilation context) and so we have to consider the effect of substituting input variables of the called module by expressions defined in the calling module. In the

```
module ModularVerificationWrapper(I,
  event ?st, ?wabrt, ?abrt, ?wsusp, ?susp)
    {
    nst: immediate await(st);
        lws: immediate weak suspend{
            lss: immediate suspend{
                immediate weak abort{
                    immediate abort{
                        M(I);
                    } when(abrt);
                } when(wabrt);
            } when(susp);
        } when(wsusp);
    }
```

Fig. 5.3: Wrapper to Define a Compilation Context

$$\begin{array}{rl}
\textbf{true} & \Rightarrow \texttt{\textbf{next(init)}} = \textbf{false}\\
\texttt{w} \wedge (\texttt{wsusp} \vee \texttt{susp}) & \Rightarrow \textbf{next}(\texttt{w}) = \textbf{true}\\
\neg(\texttt{abrt} \vee \texttt{wabrt} \vee \texttt{wsusp} \vee \texttt{susp}) \wedge (\texttt{lss} \vee \texttt{lws} \vee \texttt{st} \wedge (\texttt{nst} \vee \textbf{init}) \vee \texttt{w}) & \Rightarrow \textbf{next}(\texttt{w}) = \textbf{true}\\
(\textbf{init} \vee \texttt{nst}) \wedge \neg\texttt{st} & \Rightarrow \textbf{next}(\texttt{nst}) = \textbf{true}\\
((\textbf{init} \vee \texttt{nst}) \wedge \texttt{st} \vee \texttt{lws}) \wedge \texttt{wsusp} & \Rightarrow \textbf{next}(\texttt{lws}) = \textbf{true}\\
\texttt{lss} \wedge (\texttt{wsusp} \vee \texttt{susp}) \vee \texttt{susp} \wedge \neg\texttt{wsusp} \wedge (\texttt{lws} \vee \texttt{st} \wedge (\texttt{nst} \vee \textbf{init})) & \Rightarrow \textbf{next}(\texttt{lss}) = \textbf{true}\\
\neg(\texttt{abrt} \vee \texttt{wabrt} \vee \texttt{wsusp} \vee \texttt{susp}) \wedge (\texttt{lss} \vee \texttt{lws} \vee \texttt{st} \wedge (\texttt{nst} \vee \textbf{init}) \vee \texttt{w}) & \Rightarrow \textbf{next}(\texttt{p}) = \texttt{i}\\
\texttt{i} \wedge \neg\texttt{p} \wedge \neg\texttt{abrt} \wedge \neg\texttt{susp} \wedge (\texttt{lss} \vee \texttt{lws} \vee \texttt{st} \wedge (\texttt{nst} \vee \textbf{init}) \vee \texttt{w}) & \Rightarrow \texttt{u} = \textbf{true}\\
\texttt{p} \wedge \neg\texttt{i} \wedge \neg\texttt{abrt} \wedge \neg\texttt{susp} \wedge (\texttt{lss} \vee \texttt{lws} \vee \texttt{st} \wedge (\texttt{nst} \vee \textbf{init}) \vee \texttt{w}) & \Rightarrow \texttt{u} = \textbf{false}
\end{array}$$

Fig. 5.4: Context Interface Extension of the Open System `rfEdgeDet`

following, we consider the effect of this substitution on the state transition system: To this end, assume that formulas $\varphi_{\mathcal{I}}$ and $\varphi_{\mathcal{R}}$ over the variables $\mathcal{V}$ are given that represent the initial states and the transition relation of a state transition system $\mathcal{T}_{\mathcal{V}}$. Now consider a substitution ϱ that maps a variable $x \in \mathcal{V}$ to an expression $\varrho(x)$ over a set of variables $\mathcal{U}$. The formulas $\varrho(\varphi_{\mathcal{I}})$ and $\varrho(\varphi_{\mathcal{R}})$ then represent the following state transition system $\mathcal{T}_{\mathcal{U}} := \varrho(\mathcal{T}_{\mathcal{V}})$ over the variables $\mathcal{U}$:

- $\mathcal{S} := 2^{\mathcal{U}}$
- $\mathcal{I} := \{\vartheta \subseteq 2^{\mathcal{U}} \mid [\![\varrho(\varphi_{\mathcal{I}})]\!]_{\vartheta} = \mathsf{true}\}$
- $\mathcal{R} := \{(\vartheta, \vartheta') \subseteq 2^{\mathcal{U}} \times 2^{\mathcal{U}} \mid [\![\varrho(\varphi_{\mathcal{R}})]\!]_{\vartheta,\vartheta'} = \mathsf{true}\}$
- $\mathcal{L}(\vartheta) = \vartheta$, i.e., the state $\vartheta \subseteq 2^{\mathcal{U}}$ is the set of variables that hold in ϑ.

We first prove the following lemma:

Lemma 7 (Substitution Lemma). *Given a propositional formula φ over variables $\mathcal{V}$, a substitution ϱ that maps the variables $\mathcal{V}$ to formulas over the variables $\mathcal{U}$, and an assignment*

$\vartheta : \mathcal{U} \to \mathbb{B}$ of truth values. Then, we have $[\![\varrho(\varphi)]\!]_{\vartheta} = [\![\varphi]\!]_{\vartheta_\varrho}$ for the assignment $\vartheta_\varrho : \mathcal{V} \to \mathbb{B}$ defined by $\vartheta_\varrho(x) := [\![\varrho(x)]\!]_{\vartheta}$.

Proof. The proof is done by a simple induction on φ:

- If φ is a variable, say $\varphi \equiv x \in \mathcal{V}$, then

$$[\![\varrho(\varphi)]\!]_{\vartheta} = [\![\varrho(x)]\!]_{\vartheta} = \vartheta_\varrho(x) = [\![x]\!]_{\vartheta_\varrho} = [\![\varphi]\!]_{\vartheta_\varrho},$$

  thus the proposition holds.
- For a negation $\varphi \equiv \neg\varphi_1$, we have

$$\begin{aligned}[\![\varrho(\varphi)]\!]_{\vartheta} &= [\![\varrho(\neg\varphi_1)]\!]_{\vartheta} = [\![\neg\varrho(\varphi_1)]\!]_{\vartheta} = \neg\, [\![\varrho(\varphi_1)]\!]_{\vartheta} \\ &\overset{IH}{=} \neg\, [\![\varphi_1]\!]_{\vartheta_\varrho} = [\![\neg\varphi_1]\!]_{\vartheta_\varrho} = [\![\varphi]\!]_{\vartheta_\varrho}\end{aligned}$$

- For a conjunction $\varphi \equiv \varphi_1 \wedge \varphi_2$, we have

$$\begin{aligned}[\![\varrho(\varphi)]\!]_{\vartheta} &= [\![\varrho(\varphi_1 \wedge \varphi_2)]\!]_{\vartheta} &&= [\![\varrho(\varphi_1) \wedge \varrho(\varphi_2)]\!]_{\vartheta} \\ &= [\![\varrho(\varphi_1)]\!]_{\vartheta} \wedge [\![\varrho(\varphi_2)]\!]_{\vartheta} &&\overset{IH}{=} [\![\varphi_1]\!]_{\vartheta_\varrho} \wedge [\![\varphi_2]\!]_{\vartheta_\varrho} \\ &= [\![\varphi_1 \wedge \varphi_2]\!]_{\vartheta_\varrho} &&= [\![\varphi]\!]_{\vartheta_\varrho}\end{aligned}$$

For this reason, there is a mapping of states $\vartheta \subseteq 2^{\mathcal{U}}$ to states $\vartheta_\varrho \subseteq 2^{\mathcal{V}}$, and we define the relation $(\vartheta,\xi) \in \sigma :\Leftrightarrow \xi = \vartheta_\varrho$, and prove that this relation is somehow a simulation relation between $\mathcal{T}_{\mathcal{V}}$ and $\mathcal{T}_{\mathcal{U}}$:

Lemma 8. *Let $\mathcal{T}_{\mathcal{V}} = (\mathcal{S}_{\mathcal{V}},\mathcal{I}_{\mathcal{V}},\mathcal{R}_{\mathcal{V}},\mathcal{L}_{\mathcal{V}})$ be a transition system given by the symbolic representations of its initial states $\varphi_{\mathcal{I}}$ and its state transitions $\varphi_{\mathcal{R}}$ over the variables $\mathcal{V}$. For any substitution ϱ of the variables $\mathcal{V}$ by expressions over some set of variables $\mathcal{U}$, we define the transition system $\mathcal{T}_{\mathcal{U}} = (\mathcal{S}_{\mathcal{U}},\mathcal{I}_{\mathcal{U}},\mathcal{R}_{\mathcal{U}},\mathcal{L}_{\mathcal{U}})$ by the initial states $\varrho(\varphi_{\mathcal{I}})$ and its state transitions $\varrho(\varphi_{\mathcal{R}})$. Then, the following holds for $\mathcal{T}_{\mathcal{V}}$ and $\mathcal{T}_{\mathcal{U}}$:*

- *For every state $\vartheta \in \mathcal{S}_{\mathcal{U}}$, there is a corresponding state $\vartheta_\varrho \in \mathcal{S}_{\mathcal{V}}$ defined by $\vartheta_\varrho(x) := [\![\varrho(x)]\!]_{\vartheta}$.*
- *For every initial state $\vartheta \in \mathcal{I}_{\mathcal{U}}$, there is a corresponding initial state $\vartheta_\varrho \in \mathcal{I}_{\mathcal{V}}$ defined by $\vartheta_\varrho(x) := [\![\varrho(x)]\!]_{\vartheta}$.*
- *For every transition $(\vartheta,\vartheta') \in \mathcal{R}_{\mathcal{U}}$, there is a corresponding transition $(\vartheta_\varrho,\vartheta'_\varrho) \in \mathcal{R}_{\mathcal{V}}$.*

Proof. The first proposition is clear since for every $\vartheta \subseteq \mathcal{U}$, we can compute $\vartheta_\varrho(x) := [\![\varrho(x)]\!]_{\vartheta}$, i.e., the subset $\vartheta_\varrho \subseteq \mathcal{V}$ that satisfies $x \in \vartheta_\varrho \Leftrightarrow [\![\varrho(x)]\!]_{\vartheta} = \mathsf{true}$. For the second proposition, we have to prove that this mapping of states from $\mathcal{S}_{\mathcal{U}}$ to $\mathcal{S}_{\mathcal{V}}$ preserves membership in the initial state set, which is seen as follows:

$$\vartheta \in \mathcal{I}_{\mathcal{U}} \Leftrightarrow [\![\varrho(\varphi_{\mathcal{I}})]\!]_{\vartheta} = \mathsf{true} \Leftrightarrow [\![\varphi_{\mathcal{I}}]\!]_{\vartheta_\varrho} = \mathsf{true} \Leftrightarrow \vartheta_\varrho \in \mathcal{I}_{\mathcal{V}}$$

Similarly, we have for the third proposition:

$$\begin{aligned}(\vartheta,\vartheta') \in \mathcal{R}_{\mathcal{U}} &\Leftrightarrow [\![\varrho(\varphi_{\mathcal{R}})]\!]_{\vartheta,\vartheta'} = \mathsf{true} \\ &\Leftrightarrow [\![\varphi_{\mathcal{R}}]\!]_{\vartheta_\varrho,\vartheta'_\varrho} = \mathsf{true} \\ &\Leftrightarrow (\vartheta_\varrho,\vartheta'_\varrho) \in \mathcal{R}_{\mathcal{V}}\end{aligned}$$

We may also define an equivalence relation on the states of $\mathcal{T}_{\mathcal{U}}$: $\vartheta^1 \approx_\varrho \vartheta^2 :\Leftrightarrow \vartheta^1_\varrho = \vartheta^2_\varrho$, i.e., two states are equivalent if the evaluation of all variables with respect to ϱ matches: $\vartheta^1 \approx_\varrho \vartheta^2 :\Leftrightarrow \forall x \in \mathcal{V}.\ [\![\varrho(x)]\!]_{\vartheta^1} = [\![\varrho(x)]\!]_{\vartheta^2}$. By construction, all equivalent states are associated with the same state ϑ^i_ϱ of $\mathcal{T}_{\mathcal{V}}$, so that the quotient structure [Schn03] $\mathcal{T}_{\mathcal{U}|\approx_\varrho}$ is simulated by $\mathcal{T}_{\mathcal{V}}$. Below, we present a proof without the quotient structure construction.

Thus, if $\pi : \mathbb{N} \to \mathcal{S}_{\mathcal{U}}$ is a path in $\mathcal{T}_{\mathcal{U}}$, then there is a corresponding path π_ϱ in $\mathcal{T}_{\mathcal{V}}$ that is obtained by $\pi_\varrho(t) := (\pi(t))_\varrho$.

Theorem 3 (Preservation of Universal Specifications). *Given state transition systems $\mathcal{T}_{\mathcal{V}} = (\mathcal{S}_{\mathcal{V}}, \mathcal{I}_{\mathcal{V}}, \mathcal{R}_{\mathcal{V}}, \mathcal{L}_{\mathcal{V}})$ and $\mathcal{T}_{\mathcal{U}} = (\mathcal{S}_{\mathcal{U}}, \mathcal{I}_{\mathcal{U}}, \mathcal{R}_{\mathcal{U}}, \mathcal{L}_{\mathcal{U}})$ over variables $\mathcal{V}$ and $\mathcal{U}$, respectively, such that $\mathcal{T}_{\mathcal{V}}$ and $\mathcal{T}_{\mathcal{U}}$ are represented by the initial state conditions $\varphi_{\mathcal{I}}$ and $\varrho(\varphi_{\mathcal{I}})$ and the transition relations $\varphi_{\mathcal{R}}$ and $\varrho(\varphi_{\mathcal{R}})$, respectively (where ϱ is a substitution of variables $\mathcal{V}$ to expressions over variables $\mathcal{U}$). Then, the following holds:*

1. *for every state ϑ and every ACTL^* state formula φ, we have $(\mathcal{T}_{\mathcal{V}}, \vartheta_\varrho) \models \varphi \Rightarrow (\mathcal{T}_{\mathcal{U}}, \vartheta) \models \varrho(\varphi)$*
2. *for every path π and every ACTL^* path formula φ, we have $(\mathcal{T}_{\mathcal{V}}, \pi_\varrho) \models \varphi \Rightarrow (\mathcal{T}_{\mathcal{U}}, \pi) \models \varrho(\varphi)$*

We therefore have the following proof rules:

$$\frac{\mathcal{T}_{\mathcal{V}} \models \varphi}{\mathcal{T}_{\mathcal{U}} \models \varrho(\varphi)} \qquad\qquad \frac{\mathcal{T}_{\mathcal{V}}, \vartheta_\varrho \models \varphi}{\mathcal{T}_{\mathcal{U}}, \vartheta \models \varrho(\varphi)}$$

Proof. The proof is done by an induction on φ, where we assume without loss of generality that the formula is in negation normal form:

- If φ is a propositional formula, then

$$\begin{aligned}(\mathcal{T}_{\mathcal{V}}, \vartheta_\varrho) \models \varphi &\Leftrightarrow [\![\varphi]\!]_{\vartheta_\varrho} = \mathsf{true}\\ &\Leftrightarrow [\![\varrho(\varphi)]\!]_{\vartheta} = \mathsf{true}\\ &\Leftrightarrow (\mathcal{T}_{\mathcal{U}}, \vartheta) \models \varrho(\varphi)\end{aligned}$$

- For a path quantifier, we have

$$\begin{aligned}(\mathcal{T}_{\mathcal{V}}, \vartheta_\varrho) \models \mathsf{A}\varphi &\Leftrightarrow \text{for all paths } \pi_\varrho \text{ starting in state } \vartheta_\varrho,\\ &\ \text{we have } (\mathcal{T}_{\mathcal{V}}, \pi_\varrho) \models \varphi\\ &\overset{IH}{\Rightarrow} \text{for all paths } \pi \text{ starting in state } \vartheta,\\ &\ \text{we have } (\mathcal{T}_{\mathcal{U}}, \pi) \models \varrho(\varphi)\\ &\Leftrightarrow (\mathcal{T}_{\mathcal{U}}, \vartheta) \models \mathsf{A}\varrho(\varphi)\\ &\Leftrightarrow (\mathcal{T}_{\mathcal{U}}, \vartheta) \models \varrho(\mathsf{A}\varphi)\end{aligned}$$

  Note here that every path π in $\mathcal{T}_{\mathcal{U}}$ is the image of a path π_ϱ in $\mathcal{T}_{\mathcal{V}}$ obtained via ϱ.
- The remaining cases for state formulas $\neg\varphi$, $\varphi \wedge \psi$ and path formulas $\neg\varphi$, $\varphi \wedge \psi$, $\mathsf{X}\varphi$, $[\varphi\ \mathsf{U}\ \psi]$, and $[\varphi\ \underline{\mathsf{U}}\ \psi]$ follow directly from the induction hypothesis.

5.2 Lifting Verification Results for Pre-emption Statements

In the previous section the pre-emption context was simulated by introducing new input variables for the verification task. Hence, some assumptions about the context were made during the verification of a module. In this section, however, we lift a given verification result $(\mathcal{G},\mathcal{L}) \models \varphi$ where $\mathcal{G}$ does not consider any pre-emption statement and contains the label $\mathcal{L}$ to new results:

- $(\Theta^{\mathsf{wk}}_{\mathsf{ab}}(\mathcal{M},\sigma),\mathcal{L}) \models \Theta^{\mathsf{wk}}_{\mathsf{ab}}(\varphi,\sigma)$,
- $(\Theta^{\mathsf{st}}_{\mathsf{ab}}(\mathcal{M},\sigma),\mathcal{L}) \models \Theta^{\mathsf{st}}_{\mathsf{ab}}(\varphi,\sigma)$,
- $(\Theta^{\mathsf{wk}}_{\mathsf{sp}}(\mathcal{M},\sigma),\mathcal{L}) \models \Theta^{\mathsf{wk}}_{\mathsf{sp}}(\varphi,\sigma)$, and
- $(\Theta^{\mathsf{st}}_{\mathsf{sp}}(\mathcal{M},\sigma),\mathcal{L}) \models \Theta^{\mathsf{st}}_{\mathsf{sp}}(\varphi,\sigma)$,

where $\Theta^{\mathsf{wk}}_{\mathsf{ab}}(\mathcal{M},\sigma)$, $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\mathcal{M},\sigma)$, $\Theta^{\mathsf{wk}}_{\mathsf{sp}}(\mathcal{M},\sigma)$, $\Theta^{\mathsf{st}}_{\mathsf{sp}}(\mathcal{M},\sigma)$ are **immediate weak abort** $\mathcal{M}$ **when**(σ), **immediate abort** $\mathcal{M}$ **when**(σ), **immediate weak suspend** $\mathcal{M}$ **when**(σ), and **immediate suspend** $\mathcal{M}$ **when**(σ), respectively. Thus, concerning pre-emption statements, the results presented here are stronger since they allow us to introduce pre-emption in the module even if it has not been considered there from the beginning.
The outline of this section is as follows: first we define the pre-emptions $\Theta^{\mathsf{wk}}_{\mathsf{ab}}(\mathcal{G},\sigma)$, $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\mathcal{G},\sigma)$, $\Theta^{\mathsf{wk}}_{\mathsf{sp}}(\mathcal{G},\sigma)$, and $\Theta^{\mathsf{st}}_{\mathsf{sp}}(\mathcal{G},\sigma)$ for a set of guarded actions $\mathcal{G}$. Then, the transformations $\Theta^{\mathsf{wk}}_{\mathsf{ab}}(\varphi,\sigma)$, $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\varphi,\sigma)$, $\Theta^{\mathsf{wk}}_{\mathsf{sp}}(\varphi,\sigma)$, and $\Theta^{\mathsf{st}}_{\mathsf{sp}}(\varphi,\sigma)$ are defined and correctness proofs are given. To this end, we illustrates this approach by an example.

5.2.1 Pre-emption Statements

In the following, we describe the semantics of the four different pre-emption statements used in Quartz[2] (**immediate (weak) abort, immediate (weak) suspend**).

Definition 32 (Pre-emption of Synchronous Systems). *Given guarded actions $\mathcal{G}$ of a synchronous system over input $\mathcal{V}_i$, label $\mathcal{V}_l$, state $\mathcal{V}_s$, and output variables $\mathcal{V}_o$. Then, the weak/strong abortion and weak/strong suspension with a condition σ is obtained by modifying the guarded actions as follows to obtain synchronous systems $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\mathcal{G},\sigma)$, $\Theta^{\mathsf{wk}}_{\mathsf{ab}}(\mathcal{G},\sigma)$, $\Theta^{\mathsf{st}}_{\mathsf{sp}}(\mathcal{G},\sigma)$, and $\Theta^{\mathsf{wk}}_{\mathsf{sp}}(\mathcal{G},\sigma)$, respectively.*

pre-emption		control flow $(\gamma \Rightarrow \mathbf{next}(\ell) = \mathsf{true}) \in \mathcal{G}$	data flow $(\gamma \Rightarrow \alpha) \in \mathcal{G}$
strong abort σ	$\Theta^{\mathsf{st}}_{\mathsf{ab}}(\mathcal{G},\sigma)$	$\neg\sigma \wedge \gamma \Rightarrow \mathbf{next}(\ell) = \mathsf{true}$	$\neg\sigma \wedge \gamma \Rightarrow \alpha$
weak abort σ	$\Theta^{\mathsf{wk}}_{\mathsf{ab}}(\mathcal{G},\sigma)$	$\neg\sigma \wedge \gamma \Rightarrow \mathbf{next}(\ell) = \mathsf{true}$	$\gamma \Rightarrow \alpha$
strong suspend σ	$\Theta^{\mathsf{st}}_{\mathsf{sp}}(\mathcal{G},\sigma)$	$(\neg\sigma \wedge \gamma) \vee (\ell \wedge \sigma) \Rightarrow \mathbf{next}(\ell) = \mathsf{true}$	$\neg\sigma \wedge \gamma \Rightarrow \alpha$
weak suspend σ	$\Theta^{\mathsf{wk}}_{\mathsf{sp}}(\mathcal{G},\sigma)$	$(\neg\sigma \wedge \gamma) \vee (\ell \wedge \sigma) \Rightarrow \mathbf{next}(\ell) = \mathsf{true}$	$\gamma \Rightarrow \alpha$

The table shows that the guarded actions of the data flow are only modified by the strong pre-emption statements since weak pre-emption allows data actions to take place at the time of pre-emption. Moreover, weak and strong abortions have the same effect on the control flow. Abortion statements disable all assignments to control flow labels ℓ so that the control flow leaves the system in case of abortion. During a suspension, the control flow is kept and does not move to other labels.

[2] We only consider the **immediate** variants of these statements in this section (and will omit from now on the **immediate** keyword) that observe the pre-emption condition also in the first macro step of the statement while other variants omit the starting point of time. All results presented here can be easily transferred to the omitted delayed variants as well.

Any pre-emption context represented by the transition system $\mathcal{T}' := (\mathcal{S}', \mathcal{I}', \mathcal{R}', \mathcal{L}')$ changes the behavior only if σ holds. Hence on a path π where no pre-emption takes place ($\forall i.\pi^{(i)} \not\models \sigma$), the behavior of $\mathcal{T}'$ is equivalent to the original transition system $\mathcal{T} := (\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$. Hence, it is clear that we have $\mathcal{S} \subseteq \mathcal{S}'$, $\mathcal{I} \subseteq \mathcal{I}'$ and $\mathcal{R} \subseteq \mathcal{R}'$, which allows us to apply Lemma6.

5.2.2 Making LTL Specifications Preemptive

In general, a temporal logic formula φ that holds in a synchronous system given by its guarded actions $\mathcal{G}$ will no longer be valid in one of the systems $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\mathcal{G},\sigma)$, $\Theta^{\mathsf{wk}}_{\mathsf{ab}}(\mathcal{G},\sigma)$, $\Theta^{\mathsf{st}}_{\mathsf{sp}}(\mathcal{G},\sigma)$, and $\Theta^{\mathsf{wk}}_{\mathsf{sp}}(\mathcal{G},\sigma)$.

Example

For example, the system

$$\mathcal{G} = \left\{ \begin{array}{l} \mathsf{true} \Rightarrow \mathbf{next}(\ell) = \mathsf{true}, \\ \ell \Rightarrow c = i \end{array} \right\}$$

with $\mathcal{V}_i = \{i\}$, $\mathcal{V}_l = \{\ell\}$, and $\mathcal{V}_o = \{c\}$ is modified to

$$\Theta^{\mathsf{st}}_{\mathsf{ab}}(\mathcal{G}, \mathtt{abrt}) = \left\{ \begin{array}{l} \neg\mathtt{abrt} \Rightarrow \mathbf{next}(\ell) = \mathsf{true} \\ \neg\mathtt{abrt} \wedge \ell \Rightarrow c = i \end{array} \right\}.$$

Therefore, the LTL specification **A G** `(c↔i)` that holds on $\mathcal{G}$ is no longer satisfied in $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\mathcal{G}, \mathtt{abrt})$. However, a specification like $\mathbf{A}[(\mathtt{c} \leftrightarrow \mathtt{i})\ \mathsf{U}\ \mathtt{abrt}]$ holds, which states that `c` is equivalent to `i` until an abortion takes place.

In the following, we define transformations $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\varphi,\sigma)$, $\Theta^{\mathsf{wk}}_{\mathsf{ab}}(\varphi,\sigma)$, $\Theta^{\mathsf{st}}_{\mathsf{sp}}(\varphi,\sigma)$, and $\Theta^{\mathsf{wk}}_{\mathsf{sp}}(\varphi,\sigma)$ for temporal logic formulas φ so that we establish the following modular proof rules. These rules allow us to reason about a satisfied temporal logic property (e.g. $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\varphi,\sigma)$) of a system in a pre-emption context (e.g. $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\mathcal{G},\sigma)$), in case the property φ has already been proved for $\mathcal{G}$. Since the rules will be used in an interactive verification tool that considers systems defined by guarded actions, these rules are defined directly on guarded actions. Nevertheless, the correctness proofs will use the equivalent representation of transition systems that was defined in a previous section.

$$\frac{(\mathcal{G},\mathcal{L}) \models \varphi}{(\Theta^{\mathsf{st}}_{\mathsf{ab}}(\mathcal{G},\sigma),\mathcal{L}) \models \Theta^{\mathsf{st}}_{\mathsf{ab}}(\varphi,\sigma)} \qquad \frac{(\mathcal{G},\mathcal{L}) \models \varphi}{(\Theta^{\mathsf{wk}}_{\mathsf{ab}}(\mathcal{G},\sigma),\mathcal{L}) \models \Theta^{\mathsf{wk}}_{\mathsf{ab}}(\varphi,\sigma)}$$

$$\frac{(\mathcal{G},\mathcal{L}) \models \varphi \quad \mathsf{DFNxtEvtFree}(\mathcal{G})}{(\Theta^{\mathsf{st}}_{\mathsf{sp}}(\mathcal{G},\sigma),\mathcal{L}) \models \Theta^{\mathsf{st}}_{\mathsf{sp}}(\varphi,\sigma)} \qquad \frac{(\mathcal{G},\mathcal{L}) \models \varphi}{(\Theta^{\mathsf{wk}}_{\mathsf{sp}}(\mathcal{G},\sigma),\mathcal{L}) \models \Theta^{\mathsf{wk}}_{\mathsf{sp}}(\varphi,\sigma)}$$

The upper part defines the assumptions of the rule, the lower part defines the conclusions that hold by the rule. The condition $\mathsf{DFNxtEvtFree}(\mathcal{G})$ and the transformation $\Theta^{\mathsf{st}}_{\mathsf{sp}}(\varphi,\sigma)$ are explained in Section 5.2.4.

To this end, it is assumed without loss of generality that the given specification φ is in negation normal form and the next operators are shifted inwards such that next operators only occur in front of a variable, its negation or a next operator.

5.2.3 Transformation for Strong Abortion

An abortion can stop the execution of a system in every step. Hence, a preemptive specification should express that either the specification φ has already been satisfied or that the execution was aborted in a step before the specification was fulfilled (or violated). These thoughts lead to the following definition.

Definition 33 (Transformation $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\varphi,\sigma)$). *The transformation $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\varphi,\sigma)$ that generates an* **`abort`***-sensitive specification for $\mathsf{A}\varphi$ is defined recursively as*

$$\Theta^{\mathsf{st}}_{\mathsf{ab}}(\varphi,\sigma) := \begin{cases} \sigma \vee \varphi, & \textit{if } \varphi \textit{ is propositional} \\ \sigma \vee \mathbf{X}(\Theta^{\mathsf{st}}_{\mathsf{ab}}(\psi,\sigma)), & \textit{if } \varphi = \mathbf{X}\psi \\ [\Theta^{\mathsf{st}}_{\mathsf{ab}}(\psi,\sigma) \otimes \Theta^{\mathsf{st}}_{\mathsf{ab}}(\gamma,\sigma)], & \textit{if } \varphi = \psi \otimes \gamma \textit{ with } \otimes \in \{\underline{\mathsf{U}}, \mathsf{U}\} \\ \Theta^{\mathsf{st}}_{\mathsf{ab}}(\psi,\sigma) \otimes \Theta^{\mathsf{st}}_{\mathsf{ab}}(\gamma,\sigma), & \textit{if } \varphi = \psi \otimes \gamma \textit{ with } \otimes \in \{\wedge, \vee\}. \end{cases}$$

The crucial point of the definition is that we have to forbid the use of a variable after an abortion took place, which is achieved in that all recursive calls will finally introduce a disjunction with σ. The definition states that for the next operator, the specification $\varphi = \mathbf{X}\psi$ must lead to a specification that requires that the execution is aborted in the current or next step since σ holds or ψ holds in the next step. Thus, the specification $\varphi := [\psi \ \mathsf{U} \ \gamma]$ (and $[\psi \ \underline{\mathsf{U}} \ \gamma]$ respectively) requires that ψ holds in every step until (eventually) γ or σ holds (the condition σ is added implicitly by the recursive calls). Note that it is impossible to abort the left-hand side of a (strong) until without aborting the right-hand side, too. The same is valid for the Boolean operators because σ is added simultaneously on both sides. For a propositional formula φ, we have for example $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\mathsf{G}\varphi,\sigma) = [\varphi \ \mathsf{U} \ \sigma]$ and $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\mathsf{F}\varphi,\sigma) = \mathsf{F}(\varphi \vee \sigma)$.

Correctness

To prove the correctness of the proof rule related to the above transformation, we will make use of the following lemmata.

Lemma 9 (Containment of φ). *The transformation preserves the original specification if no pre-emption takes place, i.e., $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\varphi,\mathsf{false}) = \varphi$ holds.*

Proof. The lemma can be easily proved by induction over φ. ■

The following lemma states that the transformed specifications are vacuously satisfied if σ holds.

Lemma 10. *For an arbitrary but fixed condition σ and a path π' through $\Theta^{\mathsf{st}}_{\mathsf{ab}}(\mathcal{G},\sigma)$ and a position m such that $\pi'^{(m)} \vdash \sigma$ holds, we have*

$$(\Theta^{\mathsf{st}}_{\mathsf{ab}}(\mathcal{G},\sigma),\pi',m) \models \Theta^{\mathsf{st}}_{\mathsf{ab}}(\varphi,\sigma).$$

Proof. The proof can be easily shown by an induction over the syntax tree of φ.

The following theorem ensures the correctness of the modular proof rule for strong abortion.

Theorem 4. *For any set of guarded actions $\mathcal{G}$ with a corresponding label set $\mathcal{L}$ and any condition σ, the following holds*

$$(\mathcal{G},\mathcal{L}) \models \varphi \to (\Theta_{\mathsf{ab}}^{\mathsf{st}}(\mathcal{G},\sigma),\mathcal{L}) \models \Theta_{\mathsf{ab}}^{\mathsf{st}}(\varphi,\sigma).$$

Proof. Let $\mathcal{T}$ be the original transition system for $\mathcal{G}$ and $\mathcal{T}'$ be the transition system for $\Theta_{\mathsf{ab}}^{\mathsf{st}}(\mathcal{G},\sigma)$. Obviously, if σ does not occur on a path π' through $\mathcal{T}'$, then the original system $\mathcal{T}$ already contained π' and we can conclude from Lemma 9 that $(T',\pi') \models \Theta_{\mathsf{ab}}^{\mathsf{st}}(\varphi,\sigma) = \varphi$.

Assume we have a path $\pi \in \mathcal{T}$ through the original system and $\pi' \in \mathcal{T}'$ is a path that is equivalent to π up to a minimal position t_σ where σ holds. By induction on the number of temporal operators ($\|\varphi\|$) in an arbitrary formula φ it is shown that $\forall m \le t_\sigma$: if $(\mathcal{T},\pi,m) \models \varphi$ we have $(\mathcal{T}',\pi',m) \models \Theta_{\mathsf{ab}}^{\mathsf{st}}(\varphi,\sigma)$.

Base Case: $\|\varphi\| = 0$, hence φ is propositional and $\Theta_{\mathsf{ab}}^{\mathsf{st}}(\varphi,\sigma)$ is equivalent to $\varphi \vee \sigma$. A case distinction for $\pi'^{(m)}$ solves the case: for $\pi'^{(m)} \vdash \sigma$ we have $(\mathcal{T}',\pi',m) \models \sigma$ and for $\pi'^{(m)} \nvdash \sigma$ we have $(\mathcal{T}',\pi',m) \models \varphi$ following from the definition of π and π'. Hence, $(\mathcal{T}',\pi',m) \models \varphi \vee \sigma = \Theta_{\mathsf{ab}}^{\mathsf{st}}(\varphi,\sigma)$ holds.

Inductive Step: $\|\varphi\| = m+1$, hence, $\Theta_{\mathsf{ab}}^{\mathsf{st}}(\varphi,\sigma)$'s result is besides the trivial Boolean combinations either $\sigma \vee \mathbf{X}\Theta_{\mathsf{ab}}^{\mathsf{st}}(\psi,\sigma)$, $\left[\Theta_{\mathsf{ab}}^{\mathsf{st}}(\psi,\sigma)\ \underline{\mathsf{U}}\ \Theta_{\mathsf{ab}}^{\mathsf{st}}(\gamma,\sigma)\right]$, or $\left[\Theta_{\mathsf{ab}}^{\mathsf{st}}(\psi,\sigma)\ \mathsf{U}\ \Theta_{\mathsf{ab}}^{\mathsf{st}}(\gamma,\sigma)\right]$.
For the next operator we have $(\mathcal{T},\pi,m) \models \mathbf{X}\psi \overset{def}{\Rightarrow} (\mathcal{T},\pi,m+1) \models \psi$. If $m+1 < t_\sigma$, we can apply the inductive hypothesis to conclude $(\mathcal{T}',\pi',m+1) \models \Theta_{\mathsf{ab}}^{\mathsf{st}}(\psi,\sigma)$. Otherwise, σ holds at position $m+1$, and one can conclude from Lemma 10 that $(\mathcal{T}',\pi',m+1) \models \Theta_{\mathsf{ab}}^{\mathsf{st}}(\psi,\sigma)$
Now we turn to the strong-until-operator, i. e. we consider the case that $(\mathcal{T},\pi,m) \models [\psi\ \underline{\mathsf{U}}\ \gamma]$, hence there exists a t_γ such that $\forall m \le t' < t_\gamma.\ (\mathcal{T},\pi,t') \models \psi$ and $(\mathcal{T},\pi,t_\gamma) \models \gamma$. Hence, using the induction hypothesis for $t_\gamma < t_\sigma$ gives us $\forall m \le t' < t_\gamma.(\mathcal{T}',\pi',t') \models \Theta_{\mathsf{ab}}^{\mathsf{st}}(\psi,\sigma)$ and $(\mathcal{T}',\pi',t_\gamma) \models \Theta_{\mathsf{ab}}^{\mathsf{st}}(\gamma,\sigma)$. Then, Lemma 10 allows us to conclude $(\mathcal{T}',\pi',t_\sigma) \models \Theta_{\mathsf{ab}}^{\mathsf{st}}(\gamma,\sigma)$ and applying the induction hypothesis proves that $\forall m \le t' < t_\sigma.(\mathcal{T}',\pi',t') \models \Theta_{\mathsf{ab}}^{\mathsf{st}}(\psi,\sigma)$ holds. Hence in both cases $(\mathcal{T}',\pi',m) \models \Theta_{\mathsf{ab}}^{\mathsf{st}}([\psi\ \underline{\mathsf{U}}\ \gamma],\sigma)$ holds. The case for weak until is shown analogously. ■

5.2.4 Transformation for Strong Suspension

A suspension can postpone the current execution of the guarded actions to a later point of time. Hence, no guarded action is executed during the suspension, but the delayed assignments of the previous step still take place. The **suspend**-sensitive specification must ensure that either the execution of the system is suspended, and a violation of the specification is secondary (because no step of the original system is executed) or the next macro step of the system is executed, and as a consequence, the specification must be satisfied for this step. Note that it is possible to suspend the system infinitely often and that this case must be covered as well.

Unfortunately, the transformation defined below is not applicable if the *data flow* contains **next** assignments to *event* variables, because such an assignment may get lost during a suspension. The problem is explained in detail in Theorem 5. Hence, we exclude systems violating this requirement by adding the assumption $\mathsf{DFNxtEvtFree}(\mathcal{G})$ to the rule. This condition checks that the *data flow* is free of **next** assignments to *event* variables.

Definition 34 (Transformation $\Theta^{\mathsf{st}}_{\mathsf{sp}}(\varphi,\sigma)$). *For a given specification $\mathsf{A}\varphi$, the transformation $\Theta^{\mathsf{st}}_{\mathsf{sp}}(\varphi,\sigma)$ is defined as*

$$\Theta^{\mathsf{st}}_{\mathsf{sp}}(\varphi,\sigma) := \begin{cases} [\varphi\,\mathsf{W}\,\neg\sigma], & \textit{if } \varphi \textit{ is propositional} \\ \left[(\mathbf{X}\Theta^{\mathsf{st}}_{\mathsf{sp}}(\psi,\sigma))\,\mathsf{W}\,\neg\sigma\right], & \textit{if } \varphi = \mathbf{X}\psi \\ [\Theta^{\mathsf{st}}_{\mathsf{sp}}(\psi,\sigma) \otimes \Theta^{\mathsf{st}}_{\mathsf{sp}}(\gamma,\sigma)], & \textit{if } \varphi = \psi \otimes \gamma \textit{ with } \otimes \in \{\underline{\mathsf{U}},\mathsf{U}\} \\ \Theta^{\mathsf{st}}_{\mathsf{sp}}(\psi,\sigma) \otimes \Theta^{\mathsf{st}}_{\mathsf{sp}}(\gamma,\sigma), & \textit{if } \varphi = \psi \otimes \gamma \textit{ with } \otimes \in \{\wedge,\vee\}. \end{cases}$$

The crucial point is again that we have to forbid the use of a variable whenever the suspension takes place. Note again that all recursive calls will finally introduce a weak when operator. A module satisfying a specification $\varphi := \mathbf{X}\psi$ is suspendable in two macro steps. The definition states that the evaluation is postponed to the first point of time where σ becomes false. Thus, the specifications $\varphi := [\psi\,\mathsf{U}\,\gamma]$ (and $[\psi\,\underline{\mathsf{U}}\,\gamma]$ respectively) must lead to a specification that requires that ψ holds in every step until (eventually) γ holds or an (in)finite suspension takes place (covered by the weak when operator introduced by recursive calls).

We have $\Theta^{\mathsf{st}}_{\mathsf{sp}}(\mathsf{G}\varphi,\sigma) = \mathsf{G}\,[\varphi\,\mathsf{W}\,\neg\sigma]$ and $\Theta^{\mathsf{st}}_{\mathsf{sp}}(\mathsf{F}\varphi,\sigma) = \mathsf{F}\,[\varphi\,\mathsf{W}\,\neg\sigma]$, for any propositional φ.

An interesting fact is that an infinite suspension is equivalent to an abortion, hence only a special case of it. Hence, the transformation for abort can be also obtained from the suspension transformation.

Correctness

The following theorem ensures the correctness of the modular proof rule for strong suspension.

Theorem 5. *For any set of guarded actions $\mathcal{G}$ and corresponding label set $\mathcal{L}$, where $\mathsf{DFNxtEvtFree}(\mathcal{G})$ holds for $\mathcal{G}$ and any condition σ, we have $(\mathcal{G},\mathcal{L}) \models \varphi \rightarrow (\Theta^{\mathsf{st}}_{\mathsf{sp}}(\mathcal{G},\sigma),\mathcal{L}) \models \Theta^{\mathsf{st}}_{\mathsf{sp}}(\varphi,\sigma)$.*

Since the already proved rule for **`abort`** is a special case of the suspension rule, we only have to extend the proof of Theorem 4 at the appropriate places. We will omit this here and only describe the proof idea with help of Figure 5.5.

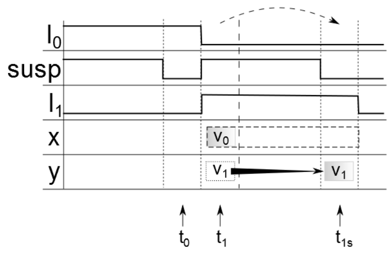


Fig. 5.5: Time Table for Suspend

```
   ⋮
l0: pause;
next (x) = v0;
l1: pause;
y = v1;
assert(φ(x,y));
   ⋮
```
Fig. 5.6: Quartz Program

The effect of a suspension on a simple Quartz program (given in Figure 5.6) is described in Figure 5.5. We consider three important points of time t_0, t_1 and t_{1s}: t_0 corresponds to a not suspended macro step starting in l_0, where the next assignment to `x` takes place. The time step t_1 is the intended execution of the macro step starting in l_1, but this step is now suspended. Nevertheless, the assignment to the variable `x` from the previous step takes place (v_0), but the immediate assignment to `y` is postponed until t_{1s}, which is the first point of time where the suspension is released. The assertion $\varphi(x,y)$ intended to be evaluated at point t_1 is postponed as well. It is no problem to evaluate $\varphi(x,y)$ in t_{1s}, since the immediate assignment is executed in the same step and for the delayed assignment the default reaction transfers the value v_0 to the step t_{1s} (indicated by the dashed box). Unfortunately, this holds only for memorized variables, since event variables are set to the type's default value and so the value v_0 gets lost during suspension. Hence, the example shows that a **`next`** assignment to an *event* variable in the data flow may completely change the behavior of the system. Hence, nothing can be deduced from the original specification. The delayed assignments to the control flow events are not problematic, i.e., are handled correctly.

5.2.5 Transformation for Weak Abortion

The weak pre-emption statements differ from their strong variants by allowing the execution of the data flow when the pre-emption takes place. If the abortion should take place at the termination point, it will therefore not modify the behavior. A **`weak abort`**-sensitive specification should express that either the specification φ is already satisfied or the execution was aborted in a state not violating the specification, but before it was ultimately fulfilled.

Definition 35 (Transformation $\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\varphi,\sigma)$). *For a given specification $\mathsf{A}\varphi$, the transformation $\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\varphi,\sigma)$ is defined as*

$$\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\varphi,\sigma) := \begin{cases} \varphi, & \textit{if } \varphi \textit{ is propositional} \\ \sigma \vee \mathbf{X}\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\psi,\sigma), & \textit{if } \varphi = \mathbf{X}\psi \\ [\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\psi,\sigma) \otimes (\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\gamma,\sigma) \vee \sigma \wedge \Theta_{\mathsf{ab}}^{\mathsf{wk}}(\psi,\sigma))], & \textit{if } \varphi = [\psi \otimes \gamma] \textit{ for } \otimes \in \{\underline{\mathsf{U}}, \mathsf{U}\} \\ \Theta_{\mathsf{ab}}^{\mathsf{wk}}(\psi,\sigma) \otimes \Theta_{\mathsf{ab}}^{\mathsf{wk}}(\gamma,\sigma), & \textit{if } \varphi = \psi \otimes \gamma \textit{ and } \otimes \in \{\wedge, \vee\} \end{cases}$$

The crucial point of the definition is that the specification must not be violated in a step where a weak abortion takes place. Hence, for the evaluation of a variable the value of σ is unimportant and only influences reads to the variable in a later step. This requires a different treatment of the until operators: Their evaluation must stop in a step where σ is satisfied. Furthermore, in such a step also one side of the operator must be satisfied. Hence, $\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\psi,\sigma) \vee \Theta_{\mathsf{ab}}^{\mathsf{wk}}(\gamma,\sigma)$ must hold, but the right-hand side of this disjunction is already covered by

$$\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\gamma,\sigma) \vee \sigma \wedge (\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\psi,\sigma) \vee \Theta_{\mathsf{ab}}^{\mathsf{wk}}(\gamma,\sigma)) = \Theta_{\mathsf{ab}}^{\mathsf{wk}}(\gamma,\sigma) \vee \sigma \wedge \Theta_{\mathsf{ab}}^{\mathsf{wk}}(\psi,\sigma)$$

and so it is enough to additionally demand $\sigma \wedge \Theta_{\mathsf{ab}}^{\mathsf{wk}}(\psi,\sigma)$ to successfully stop the evaluation of the operator. For the next operator, the specification $\varphi = \mathbf{X}\psi$ must lead to a specification that requires that the execution is aborted in the first step (without restrictions) or ψ holds in the next step (with/without abortion).

For example, we have $\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\mathsf{G}\varphi,\sigma) = [\varphi\ \mathsf{U}\ (\sigma \wedge \varphi)]$ and $\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\mathsf{F}\varphi,\sigma) = \mathsf{F}(\sigma \vee \varphi)$ for a propositional φ .

Correctness

The following theorem ensures the correctness of the modular proof rule for weak abortion.

Theorem 6. *For any set of guarded actions $\mathcal{G}$ and any condition σ, the following holds:*

$$(\mathcal{G},\mathcal{L}) \models \varphi \rightarrow (\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\mathcal{G},\sigma),\mathcal{L}) \models \Theta_{\mathsf{ab}}^{\mathsf{wk}}(\varphi,\sigma).$$

The proof is similar to the proof of Theorem 4: the used Lemma 9 is analogous for the weak abortion case, but Lemma 10 must be replaced by the following lemma:

Lemma 11. *Let $\mathcal{T}$ be the original transition system for $\mathcal{G}$ that satisfies φ and $\mathcal{T}'$ be the transition system for $\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\mathcal{G},\sigma)$. Assume we have paths $\pi \in \mathcal{T}$ and $\pi' \in \mathcal{T}'$ that is equivalent to π up to a minimal position where σ holds. For an arbitrary position m such that ${\pi'}^{(m)} \vdash \sigma$ holds, we have $(\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\mathcal{G},\sigma),\pi',m) \models \Theta_{\mathsf{ab}}^{\mathsf{wk}}(\varphi,\sigma)$.*

Proof. The proof can be made by an induction over the structure of φ and the fact, following from the definition of $\Theta_{\mathsf{ab}}^{\mathsf{wk}}$ that $(\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\mathcal{G},\sigma),\pi',m) \models \varphi$ holds.

With this lemma and the fact inferred from Definition $\Theta_{\mathsf{ab}}^{\mathsf{wk}}$ that the considered paths π and π' are equivalent up to t_σ, the proof is analogous to Theorem 4.

5.2.6 Transformation for Weak Suspension

A weak suspension freezes the control flow, but the data flow is not affected. Hence, the **`weak suspend`**-sensitive specification must express that in case of a suspension, the current state is not left which motivates the following definition.

Definition 36 (Transformation $\Theta_{\mathsf{sp}}^{\mathsf{wk}}(\varphi,\sigma)$). *Given $\Omega := \mathsf{G}(\sigma \wedge \Theta_{\mathsf{sp}}^{\mathsf{wk}}(\gamma,\sigma))$ and $\otimes \in \{\wedge,\vee,\mathsf{U}\}$, then we define*

$$\Theta_{\mathsf{sp}}^{\mathsf{wk}}(\varphi,\sigma) := \begin{cases} [(\sigma \wedge \varphi)\ \mathsf{U}\ (\neg\sigma \wedge \varphi)], & \textit{if } \varphi \textit{ is propositional} \\ \left[\sigma\ \mathsf{U}\ \neg\sigma \wedge \mathbf{X}\Theta_{\mathsf{sp}}^{\mathsf{wk}}(\psi,\sigma)\right], & \textit{if } \varphi = \mathbf{X}\psi \\ \left[\Theta_{\mathsf{sp}}^{\mathsf{wk}}(\psi,\sigma)\ \underline{\mathsf{U}}\ (\Theta_{\mathsf{sp}}^{\mathsf{wk}}(\gamma,\sigma) \vee \Omega)\right], & \textit{if } \varphi = [\psi\ \underline{\mathsf{U}}\ \gamma] \\ \Theta_{\mathsf{sp}}^{\mathsf{wk}}(\psi,\sigma) \otimes \Theta_{\mathsf{sp}}^{\mathsf{wk}}(\gamma,\sigma), & \textit{if } \varphi = \psi \otimes \gamma. \end{cases}$$

Regarding the examples, we have $\Theta_{\mathsf{sp}}^{\mathsf{wk}}(\mathsf{G}\varphi,\sigma) = \mathsf{G}\,[(\sigma \wedge \varphi\ \mathsf{U}\ (\neg\sigma \wedge \varphi)])$ and that $\Theta_{\mathsf{sp}}^{\mathsf{wk}}(\mathsf{F}\varphi,\sigma)$ is equivalent to $\mathsf{F}\,[(\sigma \wedge \varphi)\ \mathsf{U}\ (\neg\sigma \wedge \varphi \vee \mathsf{G}\sigma)]$ for a propositional φ.

It is again provable that the weak abortion is equivalent to an infinite weak suspension. The only difference to the strong case is that the weak until operator in $\Theta_{\mathsf{sp}}^{\mathsf{wk}}(\varphi,\sigma)$ is not changed, because both sides already cover the changes made in $\Theta_{\mathsf{ab}}^{\mathsf{wk}}(\varphi,\sigma)$. The term $\left[\Theta_{\mathsf{sp}}^{\mathsf{wk}}(\psi,\sigma)\ \mathsf{U}\ \Theta_{\mathsf{sp}}^{\mathsf{wk}}(\gamma,\sigma) \vee \mathsf{G}(\sigma \wedge (\Theta_{\mathsf{sp}}^{\mathsf{wk}}(\psi,\sigma) \vee \Theta_{\mathsf{sp}}^{\mathsf{wk}}(\gamma,\sigma)))\right]$ is reducible to the term $\left[\Theta_{\mathsf{sp}}^{\mathsf{wk}}(\psi,\sigma)\ \mathsf{U}\ \Theta_{\mathsf{sp}}^{\mathsf{wk}}(\gamma,\sigma)\right]$.

Correctness

The following theorem ensures the correctness of the modular proof rule for weak suspension.

Theorem 7. *For any set of guarded actions $\mathcal{G}$ and corresponding label set $\mathcal{L}$ and any condition σ, the following holds:* $(\mathcal{G},\mathcal{L}) \models \varphi \rightarrow (\Theta_{\mathsf{sp}}^{\mathsf{wk}}(\mathcal{G},\sigma),\mathcal{L}) \models \Theta_{\mathsf{sp}}^{\mathsf{wk}}(\varphi,\sigma)$.

Proof. The proof for the **`weak suspend`** case is analogous to the proof of Theorem 5, but the exclusion of delayed assignments to event variables (checked by $\mathsf{DFNxtEvtFree}(\mathcal{G})$) is not necessary, because all data flow assignments are executed in case of a weak suspension. Hence, the assignments to `y` and `x`'s take place at t_1 and $\varphi(\mathtt{x},\mathtt{y})$ can be evaluated there, too. We illustrate this situation in Figure 5.7 in analogy to Figure 5.5. Nevertheless, a set of guarded actions containing next assignments to event variables may only satisfy $\varphi(\mathtt{x},\mathtt{y})$ during suspension, since the assignment to `x` is lost after t_1.

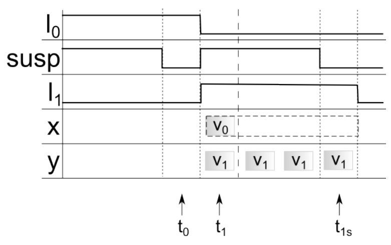


Fig. 5.7: Time Table for Weak Suspend

5.2.7 Example

In this section, the developed proof rules are applied to an example. Given an implementation of a traffic light controller, like the one represented by the (simplified) set of guarded actions in Figure 5.8 obtained from the Quartz file in Figure 5.9.

The traffic light controller has one input variable `req` and two output variables `ylw` and `grn` (indicated by `?` and `!` respectively), which are Boolean events. Thus, the outputs are **`false`** for macro steps not assigning a value to them. A traffic light usually has three lights, they will be modeled by the two output variables: `ylw=`**`true`** means that the yellow light is on, `grn=`**`true`** means that the green light is on, and `grn=`**`false`** means that the red light is on. The behavior of the controller is very simple, as long as the environment does not request a green light by `req=`**`true`**, the controller will respond by not setting any output (hence, the red light is on). A request is answered by enabling the yellow light (and the red light, since `grn=`**`false`**) in the current step, and the green light in the next step. Furthermore, it is easily provable that the controller implements the specification $\mathbf{A}\ \mathbf{G}\ (\mathtt{req} \rightarrow \mathtt{grn} \vee \mathtt{ylw} \wedge \mathbf{X}\ \mathtt{grn})$.

Assume, we want to extend the traffic light controller to operate additionally lights for crossing pedestrian (with priority). To this end, we reuse the already existing controller, like it

```
control flow:
  True ⇒¬next(init)
  ¬req∧(init∨l0) ⇒ next(l0)
  req∧(init∨l0) ⇒ next(l1)
  l1 ⇒ next(l2)
data flow:
  req∧(init∨l0∨l2) ⇒ ylw
  l1 ⇒ grn = True
specifications:
  A G (req→grn∨ylw∧(X grn))
```

Fig. 5.8: Compiled Guarded Actions

```
module TrafficLightController
  (event ?req, !ylw, !grn){
  loop{
    while (¬req){
      l0: pause;
    }
    emit (ylw);
    l1: pause;
    emit (grn);
    l2: pause;
  }
} satisfies {
 A G (req→grn∨ylw∧X grn);
}
```

Fig. 5.9: Quartz Source Code

is done in Figure 5.11[3]. In Figure 5.10, we added the guarded actions for the compiled version where we simplified the Quartz compiler's output and for a better readability, we replaced the term (`C.l0` ∨ `C.l1` ∨ `C.l2`) by `inC` and (`P.l0` ∨ `P.l1` ∨ `P.l2`) by `inP`. The original module was used twice, but embedded in two different **`abort`** statements (in the second call that models the lights for the crossing pedestrian the output for the yellow light is ignored, which is indicated by the underscore). It is not obvious that this implementation is correct, but we will see that our rules help to determine this.

```
control flow:
  True ⇒¬next(init)
  ¬reqP∧¬reqC∧(¬init∨(C.l2∨C.l0)∨inP)
    ⇒ next(C.l0)
  ¬reqP∧reqC∧(¬init∨(C.l2∨C.l0)∨inP)
    ⇒ next(C.l1)
  ¬reqP∧C.l1∧¬reqP ⇒ next(C.l2)
  reqP∧(bf∨inC∨(P.l0∨P.l2))
    ⇒ next(P.l1)
  reqP∧P.l1 ⇒ next(P.l2)
data flow:
  ¬init∧reqC∧¬reqP ⇒ ylwC
  ¬reqP∧reqC∧C.l0 ⇒ ylwC
  C.l1∧¬reqP ⇒ grnC
  reqC∧C.l2∧¬reqP ⇒ ylwC
  P.l1 ⇒ grnP
  reqP∧P.l2 ⇒P.ylw
  reqP∧(C.l0∨C.l1∨C.l2) ⇒P.ylw
  reqC∧¬reqP∧(P.l0∨P.l1∨P.l2) ⇒ ylwC
```

Fig. 5.10: Compiled Guarded Actions

```
module TrafficLightController2
  (event ?reqC, !ylwC, !grnC,
         ?reqP, !grnP,){
  loop{
    abort{
      C: TrafficLightController
          (reqC, ylwC, grnC);
    }when (reqP);
    weak abort{
      P: TrafficLightController
          (reqP, _, grnP);
    }when(¬reqP);
  }
}
```

Fig. 5.11: Quartz Source Code

Applying the rules for the two **`abort`** statements after renaming the variables in the specification φ(`req`,`ylw`,`grn`) = **`G`**(`req` → `grn` ∨ `ylw` ∧ **`X`** `grn`) leads to the following result $\Theta^{\mathsf{st}}_{\mathsf{ab}}$($\varphi$(`reqC`,`ylwC`,`grnC`),`reqP`). Hence, one has to evaluate

[3] We omitted the **`immediate`** modifier for both **`abort`** statements to be consistent with the defined rules.

$$\Theta^{\mathsf{st}}_{\mathsf{ab}}([(\neg\mathtt{reqC} \vee \mathtt{grnC} \vee \mathtt{ylwC} \wedge \mathbf{X}\ \mathtt{grnC})\ \mathsf{U}\ \mathsf{false}], \mathtt{reqP}) = \\ \mathbf{G}\ (\mathtt{reqC} \rightarrow \mathtt{reqP} \vee \mathtt{grnC} \vee \mathtt{ylwC} \wedge \mathbf{X}\ (\mathtt{grnC} \vee \mathtt{reqP}))$$

Using the same steps the specification for the **`weak abort`** can be deduced as

$$\Theta^{\mathsf{wk}}_{\mathsf{ab}}(\varphi(\mathtt{reqP},\_,\mathtt{grnP}),\sigma) = [\mathtt{reqP} \rightarrow \mathtt{grnP} \vee \mathbf{X}\ \mathtt{grnP}\ \mathsf{U}\ \neg\mathtt{reqP}]$$

.
Hence, we know that the module calls of the `TrafficLightController` together with the surrounding **`abort`** statement satisfies the corresponding specification (without having to verify it directly).

Additionally, the first specification shows that the goal of prioritizing the pedestrian's lights is reached, because `reqP` is able to shadow a green light for the cars. The second specification shows that in every step inside the second **`abort`** either $\mathtt{reqP} \rightarrow \mathtt{grnP} \vee \mathbf{X}\ \mathtt{grnP}$ or $\neg\mathtt{reqP}$ holds. Additionally, the statement before the second **`abort`** terminates if and only if `reqP` holds. Hence, in the first step of the second **`abort`** statement, the property $\mathtt{grnP} \vee \mathbf{X}\ \mathtt{grnP}$ must hold. Hence, the reuse of the traffic-light controller leads to a correct implementation.

5.3 Proof Rules for Modular Verification of Synchronous Programs

Putting the results of Section 5.1.1 and Section 5.1.2 together yields the following proof rules:

Corollary 1.

$$\frac{(\mathsf{WP}(\mathcal{T}_{\mathsf{op}}),\vartheta_\varrho) \models \mathsf{A}\varphi}{(\varrho(\mathsf{WP}(\mathcal{T}_{\mathsf{op}})),\vartheta) \models \varrho(\mathsf{A}\varphi)}$$

$$\frac{(\mathsf{WP}(\mathcal{T}_{\mathsf{op}}),\vartheta_\varrho) \models \mathsf{E}\varphi}{(\varrho(\mathsf{WP}(\mathcal{T}_{\mathsf{op}})),\vartheta) \models \varrho(\mathsf{E}\varphi)}$$

These are to be read as follows: It is assumed that a module is compiled to an open transition system with context interface $\mathsf{WP}(\mathcal{T}_{\mathsf{op}})$ and that an ACTL^* state formula $\mathsf{A}\varphi$ has been verified, i.e., for all initial states ϑ_ϱ of $\mathsf{WP}(\mathcal{T}_{\mathsf{op}})$, we have $(\mathsf{WP}(\mathcal{T}_{\mathsf{op}}),\vartheta_\varrho) \models \mathsf{A}\varphi$. This φ can be determined from the closed system with the techniques presented in Section 5.2.2 or must be given by the user. Then, we can conclude that a module call where the variables are replaced by a substitution ϱ satisfies the specification $\varrho(\mathsf{A}\varphi)$.

Note that Lemma 2 is used here also for another reason: The calling module $\mathcal{N}$ may add further actions to write variables that are now substituted for the output variables of the called module. Then, we also obtain a transition system that satisfies the assumptions of Lemma 2, so that also the additional actions of the calling module are taken into account. Hence, the fact that the calling module satisfies $\varrho(\mathsf{A}\varphi)$ is derivable from the fact that $\mathsf{WP}(\mathcal{T}_{\mathsf{op}})$ satisfies $\mathsf{A}\varphi$.

Illustrating Example

Consider again the running example `rfEdgeDet` to illustrate the use of the approach. The guarded actions for the open system of `rfEdgeDet` extended by the compilation context are listed in Figure 5.4. For this module, it is possible to verify the following specification (generated by the rules of the previous section)

$$\Theta := \mathsf{A}\,[[\Phi\ \mathsf{U}\ (\mathtt{abrt} \vee \mathtt{wabrt})]\ \mathsf{W}\ \mathtt{st}]\,,$$

where Φ is the following formula:

$$\Phi := ((\neg\mathtt{susp} \wedge \neg\mathtt{wsusp}) \rightarrow (\neg\mathtt{i} \wedge \mathsf{X}(\neg\mathtt{susp} \rightarrow (\mathtt{i} \rightarrow \mathtt{u}))))$$

The specification Θ expresses that at the first point of time where `st` holds ($[\ldots\ \mathsf{W}\ \mathtt{st}]$) property Φ will hold until an abortion takes place. Φ states that `u` holds in a step, where `i` holds and in the previous step `i` did not hold and no suspension took place.

```
module CallExample (event bool ?start, ?stop, ?i){
  event x, y;
  immediate abort {
    await (start);
    rfEdgeDet (i ⊕ x, y);
    ||
    loop{
      next (x) = i;
      if (i∧x) y = true;
      pause;
    }
  } when (stop);
}
```

Fig. 5.12: Module Call Example

The program in Figure 5.12 calls module `rfEdgeDet` with a compilation context that defines the following substitution ϱ:

$$\varrho := \left\{\begin{array}{l} (\mathtt{i}, \mathtt{i} \oplus \mathtt{x}),\\ (\mathtt{u}, \mathtt{y}),\\ (\mathtt{st}, \mathtt{start}),\\ (\mathtt{abrt}, \mathtt{stop}),\\ (\mathtt{wabrt}, \mathbf{false}),\\ (\mathtt{susp}, \mathbf{false}),\\ (\mathtt{wsusp}, \mathbf{false}) \end{array}\right\}$$

By Theorem 2 and Theorem 3, we now deduce that the system obtained by applying ϱ to (the transition system representing) the guarded actions given in Figure 5.4 satisfies the specification

$$\varrho(\Theta) = \mathsf{A}\,[[(\neg(\mathtt{i} \oplus \mathtt{x}) \wedge \mathsf{X}((\mathtt{i} \oplus \mathtt{x}) \rightarrow \mathtt{y}))\ \mathsf{U}\ \mathtt{stop}]\ \mathsf{W}\ \mathtt{start}]\,.$$

Additionally, Theorem 2 states that this specification is also satisfied for the whole module `CallExample` even though it modifies the variables used for the module call of `rfEdgeDet`.

In general, ϱ may replace different variables with the same expression, which may significantly simplify the specification. For example, the specification $\mathsf{AG}(\mathtt{a} \vee \neg \mathtt{b} \to \mathtt{c})$ will lead to $\mathsf{AG}\mathtt{c}$ by replacing the variables `a` and `b` with the same expression.

5.4 Conclusion

An approach to modularly verify synchronous systems was presented. To this end, special transition systems for open systems were constructed where the default actions on output variables are omitted, while default actions on local variables are retained. This transition system was generalized by consideration of a compilation context that takes all kinds of pre-emption statements into account. Moreover, the compilation context also provides an explicitly given start signal. Syntactically, the compilation context can be viewed as a wrapper as shown in Figure 5.3. The compilation of the module in a generalized compilation context therefore defines an open transition system $\mathsf{WP}(\mathcal{T}_{\mathsf{op}})$. If a property $\varphi \in \mathsf{ACTL}^*$ has been verified for this transition system $\mathsf{WP}(\mathcal{T}_{\mathsf{op}})$, then one can conclude that every module call that induces a substitution ϱ, satisfies the substituted formula $\varrho(\varphi)$ at the time of the module call. Candidates for these specifications can be determined by lifting verification results to pre-emption statements. Therefore, transformations was defined to modify given verification results such that these will take care of pre-emptions of the system. These transformations allow us to define modular proof rules for pre-emption statements to reason about their correctness. Thereby, it was possible to introduce pre-emption statements even though these have not been considered in the available verification results, and the transformations automatically derive new specifications that hold under the pre-emption contexts.

Chapter 6

Representation of Synchronous Systems for Verification in other MoCs

In this chapter, the representation of the synchronous MoC by the sequential/asynchronous MoC is described. Therefore, synchronous guarded actions are represented by interleaved guarded actions. Afterwards, this representation is compared with two other representations in SRI's SAL.

6.1 Translation of SGAs to Interleaved Guarded Actions

In this section, we present a translation of *SGAs* to *IGAs* to make use of tools that are based on *IGAs* for systems described by *SGAs*. The idea is to close the gap between *SGAs* and *IGAs* such that tool chains based on these different models can be connected (see Figure 6.1).

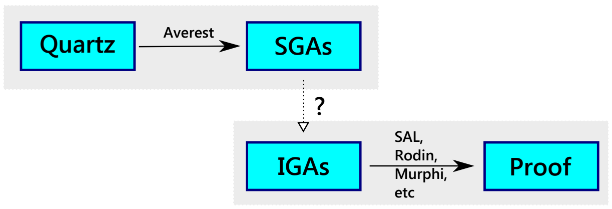


Fig. 6.1: Goal: Closing the Gap between *SGAs* and *IGAs*

Before the translation is discussed in detail, some examples will demonstrate that this translation is not trivial. The following *SGAs* are obviously enabled in every step, since their trigger conditions are always true:

$$\left\{ \begin{array}{l} \mathsf{true} \Rightarrow \mathsf{z{=}y} \\ \mathsf{true} \Rightarrow \mathsf{y{=}x} \end{array} \right\}$$

The behavior executing both *SGAs* synchronously results in a state were the three variables `x`, `y`, and `z` have the same value. Hence, starting with the valuation `x=1`, `y=0` and `z=0` as given in Figure 6.2, the variables have the values `x=1`, `y=1`, and `z=1` in the next step. Executing the *SGAs* as *IGAs* instead, shows that there is one interleaving schedule executing every guarded

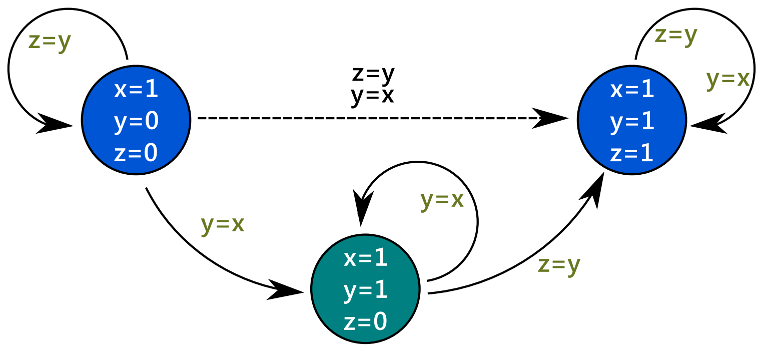


Fig. 6.2: Example 1: Execution Trace

action once to reach the target state, but there exists other schedules that require more than one execution of all guarded actions. Another example, this time with delayed assignments is:

$$(\mathsf{true} \Rightarrow \texttt{\textbf{next}(x)=y})$$
$$(\mathsf{true} \Rightarrow \texttt{\textbf{next}(y)=x})$$

The behavior executing both *SGAs* synchronously results in swapping the values of the variables `x` and `y` in each step. Hence, starting with the valuation `x=1` and `y=0` as given in Figure 6.3, the variables have the values `x=0` and `y=1` in the next step. Executing the *SGAs* as *IGAs* instead, shows that there is no interleaving schedule with the same behavior. The execution of (true ⇒ **`next`**`(x)=y`) leads to the intermediate state where `x=y=0` holds and the remaining action does not change the state further. A similar situation occurs when executing (true ⇒ **`next`**`(y)=x`), where the state `x=y=1` is obtained.

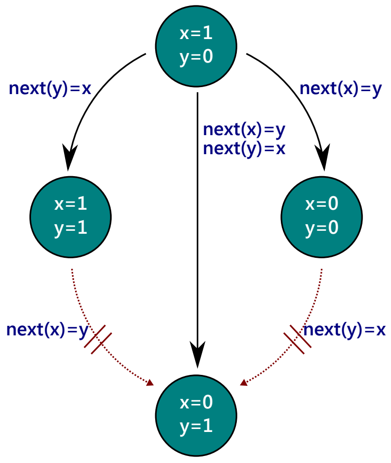


Fig. 6.3: Example 2: Execution Trace

The last example shows a combination of delayed and immediate assignments and additionally shows that the interleaved execution might have different schedules leading to the desired result. These different schedules are eliminated for the C-Synthesis by choosing one of them.

$$\left\{\begin{array}{l}\mathsf{true} \Rightarrow \mathtt{x=z} \\ \mathsf{true} \Rightarrow \mathtt{y=\neg z} \\ \mathsf{true} \Rightarrow \mathbf{next}\mathtt{(z)=\neg z}\end{array}\right\}$$

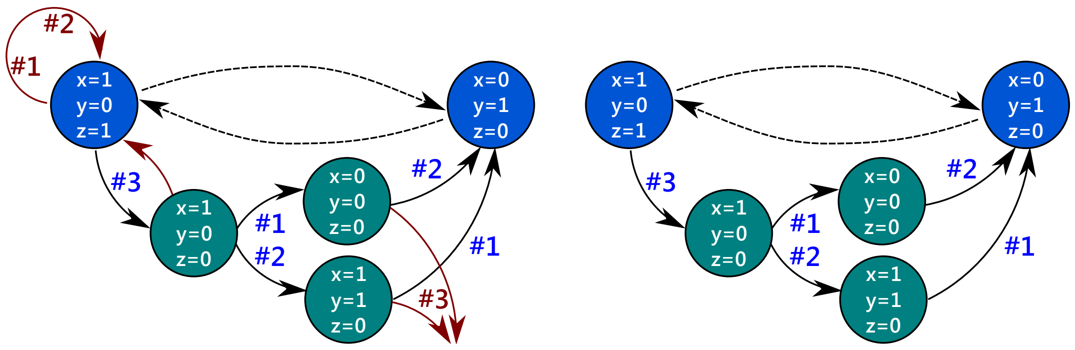


Fig. 6.4: Example 3: Execution Trace

For a translation from *SGAs* to *IGAs*, four major problems have to be solved:

- *execution order*: the execution of the *IGAs* must follow the causal ordering defined by the *SGAs*
- *immediate/delayed assignments*: the different behavior of immediate and delayed assignments of *SGAs* must be preserved by *IGAs*
- *macro steps*: all *IGAs* related to the current reaction step must be executed exactly once before the *IGAs* for the next step are executed
- *temporal behavior*: the translation to *IGAs* introduces additional intermediate states that typically violate temporal logic specifications given for *SGAs* so that these specifications have to be adequately adapted.

This section is organized as follows: the following section motivates the problem and defines *IGAs*. The main part is presented in Section 6.1.2, where solutions to the identified problems are shown. In Section 6.1.8, the transformation of *SGAs* to *IGAs* is used to verify some conditions of a running example. Finally, we conclude the section and discuss future work.

Example

We will use the synchronous language Quartz to generate sets of *SGAs*.
The program given in Figure 6.5 and the corresponding *SGAs* (given in Figure 6.7) are used as a running example throughout this section and will also show that the presented transformation is not a simple sequentialization (as used, e.g., in the synthesis of C programs in the Averest system). In particular, data-independent *SGAs* can be executed in any order.

```
macro N = 5;

module example(
  event  ?mode,
  int{N} ?a,?b,?c,?d,
  int{4*N*N+1} !r) {

int{4*N*N+1} s1,s2,m;
  loop {
    if (mode){
      //r = (a + b*c + d)
      r = s2;
      s2 = s1 + d;
      s1 = a + m;
      m = b * c;
    } else {
      //r = (a + b)*(c + d)
      r = m;
      s2 = c + d;
      s1 = a + b;
      m = s1 * s2;
    }
    l: pause;
  }
} satisfies {
 a: assert A G
  (mode→(r=(a + b * c + d)));
 b: assert A G
  (¬mode→(r=(a + b)*(c + d)));
}
```

Fig. 6.5: Quartz Example

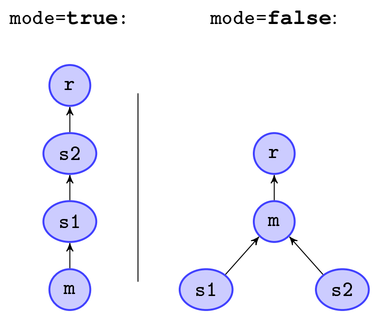


Fig. 6.6: Data Dependency Graphs

The program has five inputs (denoted by the question mark) and produces the output `r` (denoted by the exclamation mark). The type **`int`**`{n}` defines the range of a variable to the interval $[-n,\ldots,n-1]$. In each step, the output for the next step is computed depending on the input `mode` either by `a + b*c + d` (`mode=`**`true`**) or `(a + b) * (c + d)` (`mode=`**`false`**). This computation is done by using the three local variables `s1`, `s2`, and `m`. An important point is that the data dependencies of these variables are determined by the input variable `mode` as shown in Figure 6.6. The obtained *SGAs* for this program are given in Figure 6.7.

6.1.1 Interleaved Guarded Actions

As already mentioned, many tools and languages are based on models of computations that can be represented by interleaved guarded actions:

```
control flow:
  True ⇒ next(init) = False
  (l ∨ init) ⇒ next(l) = True
data flow:
  mode ∧ (l ∨ init) ⇒ r = s2
  mode ∧ (l ∨ init) ⇒ s1 = a+m
  mode ∧ (l ∨ init) ⇒ s2 = s1+d
  mode ∧ (l ∨ init) ⇒ m = b∗c
  ¬mode ∧ (l ∨ init) ⇒ r = m
  ¬mode ∧ (l ∨ init) ⇒ s2 = c+d
  ¬mode ∧ (l ∨ init) ⇒ s1 = a+b
  ¬mode ∧ (l ∨ init) ⇒ m = s1∗s2
```

Fig. 6.7: Example's Guarded Actions

Definition 37 (Interleaved Guarded Actions (*IGAs*)). *An* IGA $(\gamma \Rightarrow \mathcal{S})$ *consists of a trigger* γ *and a set of assignments* $\mathcal{S}$*. In every step, a single enabled* IGA *is nondeterministically chosen and executed. A set of* IGAs *is executed until all* IGAs *are disabled.*

The main difference between *SGAs* and *IGAs* is that all *SGAs* are executed in parallel and may thereby directly influence each other. On the other hand, *IGAs* are executed one after the other in an arbitrary nondeterministic execution order. Hence, a system description using *IGAs* is in general nondeterministic. This is not only the case for the execution order, but also for the final result in case the *IGAs* are not confluent.

It is very important for the proposed transformation that the determinism of *SGAs* is preserved even though the execution order of the generated *IGAs* is nondeterministic. This is done by introducing some restriction to maintain a confluent behavior, but we want to avoid a complete sequentialization for the obtained *IGAs.* The approach presented here allows us to execute *all* enabled IGAs in parallel. Hence, the transformation presented here allows us to target the sequential MoC (by choosing a single enabled IGA) and the concurrent MoC (by choosing a subset of the enabled IGAs).

6.1.2 Translating *SGAs* to *IGAs*

In this section, the proposed transformation is presented by solving the four major problems related to the execution order within one and between successive macro steps, the preservation of the different assignments, and the lifting of specifications from *SGAs* (macro steps) to *IGAs* (micro steps).

In Section 6.1.3, a transformation $\Gamma : \mathcal{S} \to \mathcal{I}$ from *SGAs* $\mathcal{S}$ to *IGAs* $\mathcal{I}$ is defined and its correctness is proven. The transformation is based on two phases, where the first three major problems (*execution order*, *immediate/delayed assignments*, and *macro step behavior*) are solved. Each *SGA* with an immediate assignment will be used to form an *IGA* besides an additional *IGA* for the remaining *SGAs.* In the following, we will introduce some assumptions to simplify the correctness proof. We will focus on these assumptions in Section 6.1.5, where we describe some implementation details and improvements. Afterwards, the translation of the running example is given in Section 6.1.6. The remaining major problem (*temporal specifications*) will be considered in Section 6.1.7.

6.1.3 Transformation from *SGAs* to *IGAs*

The definition of the transformation from *SGAs* to *IGAs* is based on the following assumptions.

- We consider only causally correct sets of *SGAs*. This means that the value of a single variable must not be influenced by more than one *SGA* within the same macro step. This is ensured during compilation by the causality analysis.
- We assume that the *SGAs* define the value of each variable in each step for the sake of simplicity. This means that the default reaction that determines a value in case no assignment to a variable is executed is contained explicitly in the *SGAs*. This is no real restriction, since compilers can add the default reaction explicitly. Nevertheless, we will explain later an improved transformation that does not need this assumption.
- We extend each type by an additional value $\perp$ indicating that the value is not yet determined in this macro step. For the implementation, this is done by adding a boolean flag for each variable that is set during an assignment and reset at the beginning of each macro step. In this way, the flag indicates that the value of the corresponding variable is present/valid.
- We assume for this section that the environment is responsible for changing the input values at the appropriate points of time and keeping them constant during the 'simulation' of a macro step. For the implementation, this could be ensured by introducing a local variable for each input that is only changed at the end/beginning of a macro step.

Furthermore, we split the 'simulation' of a macro step into two phases, because delayed assignments do not influence the current macro step. Instead, they only evaluate the value in the current environment and assign this value to the variable in the next step. Hence, delayed assignments could be collected first and executed later once all values of the current step are determined. Thus, we will split the set of *SGAs* $\mathcal{S}$ into the sets $\mathcal{S}_{\mathcal{I}}$ and $\mathcal{S}_{\mathcal{D}}$ that contain immediate and delayed actions, respectively. Following this, a macro step is simulated/executed in two phases by *IGAs*: (1) the current environment is computed by evaluating $\mathcal{S}_{\mathcal{I}}$, then (2) the obtained environment is used to set up a partial environment for the next step by evaluating $\mathcal{S}_{\mathcal{D}}$ and the read inputs of the next step. This is similar to the definition of reaction- and transition rules of SOS rules.

This partitioning ensures the preservation of the behavior for the two kinds of assignments (*immediate/delayed assignments*). Furthermore, the first phase handles the dependencies within a single macro step by defining a transformation for the *SGAs* only containing immediate assignments (*execution order*). Then, the second phase ensures a proper execution of macro steps (*macro step behavior*).

Phase 1: Computation of the Current Environment

This phase ensures that the generated *IGAs* are executed by respecting their data dependencies and that all read accesses to a variable will return the same value during a macro step. Additionally, it must be ensured that each enabled *IGA* is executed exactly once during a macro step (otherwise, a termination of a macro step would not be guaranteed). Since only immediate assignments directly influence the current macro step, we will only consider *SGAs* containing immediate assignments ($\mathcal{S}_{\mathcal{I}}$) in this phase.

We assume that at the beginning of each macro step, we are given an environment $\mathcal{E}$ that assigns each variable $v \in \mathcal{V}$ a value $\mathcal{E}(v)$. More precisely, inputs have a value distinct from $\perp$, all variables targeted by a delayed assignment of the previous step must have the values determined in that previous step, and all other variables must have the unknown value $\perp$.

Execution Order

The deterministic execution of *SGAs* is guaranteed by evaluating/executing them along their data dependencies, although they are executed synchronously. Hence, the transformation must ensure that all required values are known (i.e., $\neq \perp$) before an *IGA* is executed. This is achieved by adding the condition $v \neq \perp$ for each variable read in the *SGA*. In this way, all values required for the execution are known for an enabled action, and a correct *execution order* is ensured.

Monotonicity

Since an *IGA* must not be executed more than once in a macro step, the *IGAs* must be deactivated after an execution in a macro step. This is done by requiring that the target variable $\mathtt{x}$ of an *SGA* $(\gamma \Rightarrow \mathtt{x} = \tau)$ does not yet have a (valid) value by adding the condition $\mathtt{x} = \perp$. Hence, the transformation of a single *SGA* is defined as:

Definition 38. *Immediate Transformation $\mathit{\Gamma}_{\mathcal{I}}$*

$$\begin{aligned}&\mathit{\Gamma}_{\mathcal{I}}(\gamma \Rightarrow x = \tau) := \\ &\left(\Big(\bigwedge_{v \in \mathsf{read}(\gamma \Rightarrow \mathsf{x} = \tau)} v \neq \perp \Big) \wedge (x = \perp) \wedge \gamma \Rightarrow \{x = \tau\} \right),\end{aligned}$$

where $\mathsf{read}(\gamma \Rightarrow \mathsf{x} = \tau) := \mathsf{Vars}(\gamma) \cup \mathsf{Vars}(\tau)$ *and* $\{x = \tau\}$ *represents a set with a single element.*

Phase 2: Stepwise Execution

The modifications defined above ensure the (correct) data-dependent execution of a single macro step and define a complete environment $\mathcal{E}$. Finally, all triggers will become false, because all variables have a value and the secondly introduced condition will therefore deactivate all *IGAs*. Hence, we need an additional *IGA* that is activated once all others are executed to set up the new macro step and to execute the delayed assignments of $\mathcal{S}_{\mathcal{D}}$. This is done by the following *IGA* that will be called *conclusion*.

For the definition of the *conclusion*, we assume that the set $\mathcal{V}$ represents all variables contained in the set of *SGAs*, and that the function get returns for all input variables $\mathcal{V}_i \subseteq \mathcal{V}$ the value for the next macro step. Moreover, we take care of variables assigned by a delayed assignment: we consider the value τ of an enabled delayed assignment, and for all other variables, we use the value $\perp$:

$$\mathsf{set}_{\mathcal{G}}(x) := \begin{cases} \mathsf{get}(x), & \text{if } \mathtt{x} \in \mathcal{V}_i \\ \tau, & \text{if } \exists(\gamma \Rightarrow \mathbf{next}\mathtt{(x)=}\tau) \in \mathcal{G} : \mathcal{E}(\gamma) = \mathsf{true} \\ \bot, & \text{else} \end{cases}$$

$$\varGamma_{\mathcal{D}}(\mathcal{S}_{\mathcal{D}}) := \left(\bigwedge_{v \in \mathcal{V}} v \neq \bot \right) \Rightarrow \bigcup_{x \in \mathcal{V}} \left\{ x = \mathsf{set}_{\mathcal{S}_{\mathcal{D}}}(x) \right\}$$

The state where the *conclusion* is executed marks the end of the current macro step. For a correct *macro step behavior*, this state must be equivalent (all variables must have the same value) to the state reached by executing the *SGAs*, and it is the point where the environment must update the inputs.

Hence, given a set of *SGAs* $\mathcal{S}$, the transformation $\varGamma$ from $\mathcal{S}$ to a set of *IGAs* $\mathcal{I}$ is defined as follows:

Definition 39. SGA *Transformation*

$$\varGamma(\mathcal{S}) := \left(\bigcup_{g \in \mathcal{S}_{\mathcal{I}}} \{\varGamma_{\mathcal{I}}(g)\} \right) \cup \{\varGamma_{\mathcal{D}}(\mathcal{S}_{\mathcal{D}})\},$$

where the set $\mathcal{S} = \mathcal{S}_{\mathcal{I}} \cup \mathcal{S}_{\mathcal{D}}$ *is composed of* SGAs *with immediate assignments* $\mathcal{S}_{\mathcal{I}}$ *and delayed assignments* $\mathcal{S}_{\mathcal{D}}$*.*

6.1.4 Correctness of the Transformation $\varGamma$

In this section, we will prove the correctness of the presented transformation $\varGamma$, i.e., we want to show that an arbitrary macro step following a correct execution of the previous steps is determined by the generated *IGAs* in the same way as the *SGAs* define it. Therefore, we have given the input assignment $\mathcal{E}_{\mathcal{V}_i}$ for the current step that maps any input variable $v \in \mathcal{V}_i$ to a (known) value. Additionally, we know that the previous step was executed correctly, hence the set of delayed assignments $\mathcal{E}_{\mathcal{V}_d}$ that are executed in the previous step must be given. Hence, $\mathcal{E}_{\mathcal{V}_d}$ maps a variable $v \in \mathcal{V}_d$ to a (known) value. We have to show that the evaluation w.r.t. $\mathcal{E} := \mathcal{E}_{\mathcal{V}_i} \cup \mathcal{E}_{\mathcal{V}_d}$ of the *IGAs* $\mathcal{I}$ (denoted by $\mathsf{eval}_{\mathcal{I}}$) is equivalent to the evaluation of the guarded actions $\mathcal{S}$ (denoted by $\mathsf{eval}_{\mathcal{S}}$). The transformation $\varGamma_{\mathcal{D}}$ ensures with the guard $\bigwedge_{v \in \mathcal{V}} v \neq \bot$ that the set $\mathcal{S}_{\mathcal{I}}$ is evaluated before the evaluation of $\mathcal{S}_{\mathcal{D}}$. Hence, we first show that $\mathcal{S}_{\mathcal{I}}$ is executed correctly. Afterwards, we are able to use the environment defined by the execution of $\mathcal{S}_{\mathcal{I}}$ to show that $\mathcal{S}_{\mathcal{D}}$ is executed correctly.

Correct Execution of $\mathcal{S}_{\mathcal{I}}$

Here, we will show the correct execution of the set $\mathcal{S}_{\mathcal{I}}$. Therefore, we will define a partial order on the guarded actions to simplify the correctness proof. As already mentioned, the synchronous computation model demands that for a given input assignment, the values of all variables must be uniquely determined. Hence, there exists for a given input assignment a partial order of the *SGAs*. In the following, we will define the partial order *rank* w.r.t. the given input and delayed assignments $\mathcal{E}$ over the set $\mathcal{S}_{\mathcal{I}}$. Therefore, the following functions are used:

- The substitution $\langle\varphi\rangle_{\mathcal{V}}^{\mathcal{U}}$ replaces all occurrences of a variable $\mathsf{v} \in \mathcal{V}$ in the boolean expression φ by the corresponding $\mathsf{u} \in \mathcal{U}$ and simplifies the result afterwards.
- The function $\mathsf{writes}(\mathcal{G}) := \{x \mid (\gamma \Rightarrow x = \tau) \in \mathcal{G}\}$ determines the targets of a set of guarded actions.
- The function $\mathsf{read}(\gamma \Rightarrow \mathsf{x} = \tau) := \mathsf{Vars}(\gamma) \cup \mathsf{Vars}(\tau)$ returns all variables read by a guarded action.

Definition 40. *Rank: Given a set of variables $\mathcal{U}$ and a set of corresponding values $\mathcal{T}$, the* rank *of a* SGA *is defined by:*

$$\begin{aligned}
&g := (\gamma \Rightarrow \alpha) \in \mathsf{rank}_{(\mathcal{U}\times\mathcal{T})}(i) :\Leftrightarrow \\
&\langle\gamma\rangle_{\mathcal{U}}^{\mathcal{T}} \neq \mathsf{false} \wedge g \notin \bigcup_{j<i} \mathsf{rank}_{(\mathcal{U}\times\mathcal{T})}(j) \wedge \\
&\mathsf{read}\,(\gamma \Rightarrow \alpha) \subseteq \left(\mathcal{U} \cup \bigcup_{j<i} \mathsf{writes}(\mathsf{rank}_{(\mathcal{U}\times\mathcal{T})}(j))\right)
\end{aligned}$$

Given a valuation $\mathcal{U} \times \mathcal{T}$ of some variables $\mathcal{V}$ (like $\mathcal{E}$) that maps a variable $u \in \mathcal{U}$ to its value $v \in \mathcal{T}$. The *rank* describes that a *SGA* of $\mathsf{rank}_{(\mathcal{U}\times\mathcal{T})}(i)$ depends only on variables of $\mathcal{U}$ or variables written in a lower rank. In other words, the *SGAs* of $\mathsf{rank}_{(\mathcal{U}\times\mathcal{T})}(0)$ only depend on variables whose values are determined by the valuation $\mathcal{U} \times \mathcal{T}$. Hence, these do not contain unknown values. Then $\mathsf{rank}_{(\mathcal{U}\times\mathcal{T})}(1)$ depends only on variables written in $\mathsf{rank}_{(\mathcal{U}\times\mathcal{T})}(0)$ or by the valuation $\mathcal{U} \times \mathcal{T}$. There exists also a relation to the causality analysis: the action of a guarded action $s \in \mathsf{rank}_{(\mathcal{U}\times\mathcal{T})}(i)$ contributes to the computed environment in the $i-1$ step of the causality analysis.

Proof

We show by finite induction over the rank n of the *SGAs* $\mathcal{S}_{\mathcal{I}}$ that the following holds:

$$\forall s \in \mathsf{rank}_{\mathcal{E}}(n) : \mathsf{eval}_{\mathcal{S}_{\mathcal{I}}}(s) = \mathsf{eval}_{\mathcal{I}}(\Gamma_{\mathcal{I}}(s))$$

Base Case

$n = 0$, hence $s = (\gamma_s \Rightarrow \alpha_s) \in \mathsf{rank}_{\mathcal{E}}(0)$ is given. $s \in \mathcal{S}_{\mathcal{I}}$ states that α_s contains an immediate assignment $x = \tau$, and that there exists an *IGA* $i = (\gamma_i \Rightarrow \alpha_i) = \Gamma_{\mathcal{I}}(s)$. From the definition of the transformation $\Gamma_{\mathcal{I}}$, we know that α_i is a set containing α_s and $\gamma_i := \beta \wedge \gamma_i$ with $\beta = (\bigwedge_{v \in \mathsf{read(s)}} v \neq \bot) \wedge (x = \bot)$. Furthermore, $s \in \mathsf{rank}_{\mathcal{E}}(0)$ implies that $\mathsf{read}(s)$ is a subset of the variables in $\mathcal{E}$, hence all the conjuncts $v \neq \bot$ in β are true and $x = \bot$ remains, which holds before execution of i. Hence, $\mathsf{eval}_{\mathcal{S}_{\mathcal{I}}}(s) = \mathsf{eval}_{\mathcal{I}}(\Gamma_{\mathcal{I}}(s))$ holds for the base case.

Inductive Step

We know that $\mathsf{rank}_{\mathcal{E}}(n)$ is executed correctly, hence we know that $\mathsf{eval}_{\mathcal{S}_{\mathcal{I}}}(v) = \mathsf{eval}_{\mathcal{I}}(v)$ for a variable written in a rank less than or equal to n or contained in the valuation $\mathcal{E}$. We have to show that $\mathsf{rank}_{\mathcal{E}}(n+1)$ is executed correctly.

Let $s = (\gamma_s \Rightarrow \alpha_s) \in \mathsf{rank}_{\mathcal{E}}(n+1)$ be an arbitrary *SGA* of rank $n+1$ and $i = (\gamma_i \Rightarrow \alpha_i) = \Gamma_{\mathcal{I}}(s)$ be the corresponding *IGA*. From the definition of the transformation $\Gamma_{\mathcal{I}}$, we know that α_i is a set containing α_s and $\gamma_i = \beta \wedge \gamma_s$ with $\beta = (\bigwedge_{v \in \mathsf{read(s)}} v \neq \bot) \wedge (x = \bot)$. A case distinction over the evaluation of the trigger γ_s and γ_i solves the problem:

- For the case $\mathsf{eval}_{\mathcal{S}_\mathcal{I}}(\gamma_s) = \mathsf{eval}_{\mathcal{I}}(\gamma_i) = \mathsf{false}$, s and i are not enabled and both are 'executed' in the same way.
- For the case $\mathsf{eval}_{\mathcal{S}_\mathcal{I}}(\gamma_s) = \mathsf{eval}_{\mathcal{I}}(\gamma_i) = \mathsf{true}$, s and i are executed since α_i and α_s are equivalent and only contain variables of the lower ranks with equal values (induction hypothesis). Hence, both are executed correctly and $\mathsf{eval}_{\mathcal{S}_\mathcal{I}}(s) = \mathsf{eval}_{\mathcal{I}}(i)$ holds.
- The case $\mathsf{eval}_{\mathcal{S}_\mathcal{I}}(\gamma_s) = \mathsf{false}$ and $\mathsf{eval}_{\mathcal{I}}(\gamma_i) = \mathsf{true}$ is not possible, since γ_s appears as a conjunct in $\gamma_i = \beta \wedge \gamma_s$, hence the latter cannot evaluate to true if the former evaluates to false.
- For the case $\mathsf{eval}_{\mathcal{S}_\mathcal{I}}(\gamma_s) = \mathsf{true}$ and $\mathsf{eval}_{\mathcal{I}}(\gamma_i) = \mathsf{false}$, we know from $\gamma_i := \beta \wedge \gamma_s$ that
$$\beta := \left((\bigwedge_{v \in \mathsf{read(s)}} v \neq \bot) \wedge (x = \bot) \right) = \mathsf{false}$$
holds. Hence, either there exists a variable $v \in \mathsf{read}(s)$ with $v = \bot$ or $x \neq \bot$ holds. Both cases lead to a contradiction:
  - if there exists a $v \in \mathsf{read}(s)$ with $v = \bot$, then we know from the induction hypothesis that in the lower rank no *SGA* has written v and v is not contained in the variables $\mathcal{U}$ of $\mathcal{E} := (\mathcal{U} \times \mathcal{T})$. This leads to a contradiction of $s \in \mathsf{rank}_{\mathcal{E}}(n)$, since $v \in \mathsf{read}(s)$ but $v \notin \left(\mathcal{U} \cup \bigcup_{j<n} \mathsf{writes}(\mathsf{rank}_{(\mathcal{U} \times \mathcal{T})}(j)) \right)$.
  - $x \neq \bot$ implies that x was already determined by an *IGA* corresponding to a *SGA* in a lower rank, hence either a single variable is written by two different *SGAs* in a macro step (which is forbidden by the synchronous computation model) or a lower rank was incorrectly executed, which contradicts again the induction hypothesis. ■

Correct Execution of $\mathcal{S}_\mathcal{D}$

Hence, the set of all *IGAs* generated from $\mathcal{S}_\mathcal{I}$ behave like the execution of $\mathcal{S}_\mathcal{I}$ in the synchronous MoC. Additionally, all *SGAs* containing immediate assignments are executed correctly. Hence, for all variables $v \in \mathcal{V}$, we have $\mathsf{eval}_{\mathcal{S}_\mathcal{I}}(v) = \mathsf{eval}_{\mathcal{I}}(v) \neq \bot$ and the *conclusion*'s trigger defined by $\Gamma_\mathcal{D}$ is the only fulfilled one. The function call $\mathsf{set}_{\mathcal{S}_\mathcal{D}}$ returns in that case for the inputs the values for the next step, for variables with enabled guarded action $s \in \mathcal{S}_\mathcal{D}$ the correct value, since $\mathsf{eval}_{\mathcal{S}_\mathcal{I}}(s) = \mathsf{eval}_{\mathcal{I}}(\Gamma_{\mathcal{I}}(s))$ holds. For all other variables, the unknown value $\bot$ is returned. Hence, the preconditions for the next step are fulfilled and the current step was executed correctly. ■

Hence, the defined transformation is a confluent translation from *SGAs* to *IGAs*. The choice, which of the enabled *IGA* is chosen is not important; the execution of each macro step leads to the same deterministic result as defined by the *SGAs* independent of that (non-deterministic) choice.

6.1.5 Implementation Details

Before we show the translation of the running example, we will discuss some implementation details. The transformation Γ defined in Section 6.1.4 shows the correctness of the approach, but is based on several assumptions that are not practical.

Encoding Unknown Values

The extension of all data types by the unknown value $\perp$ leads to the problem that the targeted tool must either contain such a (special) data type or support user-defined data types, which must be implemented additionally. Another possibility is to encode the unknown value $\perp$ of a variable v by an additional Boolean variable v_{v}, the so-called valid flag. Hence, the value of a variable v is then defined by the tuple (v, v_{v}):

$$\mathsf{value}((v, v_{\mathrm{v}})) := \begin{cases} v, & \text{if } v_{\mathrm{v}} = \mathsf{true} \\ \perp, & \text{else} \end{cases}$$

This allows the use of typical built-in data types and only requires that the new variable v_{v} is set at the same time an assignment to v is executed.

Implementing Default Reactions

Another improvement consists of a special treatment of the default reaction. The default reaction must be executed in case no other *SGA* defines the value of a variable. This reaction could be added explicitly to the set of *SGAs*, but this may require the introduction of so-called carrier variables to store the values of the previous step if these are required for the default reaction (see [Schn09]). Using the following approach can avoid this overhead. We describe again the general idea of this approach for a single variable and discuss then some special cases and the impact on the transformation.

$$\begin{array}{lllll|lllll}
\gamma_1 & \Rightarrow & \mathtt{x} & = & \tau_1 & \delta_1 & \Rightarrow & \mathbf{next}(\mathtt{x}) & = & \upsilon_1 \\
 & \vdots & & & & & \vdots & & & \\
\gamma_n & \Rightarrow & \mathtt{x} & = & \tau_n & \delta_m & \Rightarrow & \mathbf{next}(\mathtt{x}) & = & \upsilon_m
\end{array}$$

Fig. 6.8: Guarded Actions for variable `x`

General Case

The general idea of translating *SGAs* of a single variable `x` given in Figure 6.8 is described in the following. An corresponding variable $\mathtt{x}_{\mathrm{v}}$ must be introduced to determine the validity of the value contained in `x`. Then, the *SGAs* with immediate assignments $s_1, \ldots, s_n$ are converted into:

$$
\begin{array}{lcl}
\neg \mathtt{x}_{\mathrm{v}} \wedge \left(\bigwedge\limits_{v \in \mathsf{read}(s_1)} v_{\mathrm{v}} \right) \wedge \gamma_1 & \Rightarrow & \begin{bmatrix} \mathtt{x} & = \tau_1 \\ \mathtt{x}_{\mathrm{v}} & = \mathsf{true} \end{bmatrix} \\
 & \vdots & \\
\neg \mathtt{x}_{\mathrm{v}} \wedge \left(\bigwedge\limits_{v \in \mathsf{read}(s_n)} v_{\mathrm{v}} \right) \wedge \gamma_n & \Rightarrow & \begin{bmatrix} \mathtt{x} & = \tau_n \\ \mathtt{x}_{\mathrm{v}} & = \mathsf{true} \end{bmatrix} \\
\neg \mathtt{x}_{\mathrm{v}} \wedge \left(\bigwedge\limits_{i=1\ldots n} \neg \gamma_i \right) & \Rightarrow & \begin{bmatrix} \mathtt{x}_{\mathrm{v}} & = \mathsf{true} \end{bmatrix}
\end{array}
$$

Thereby, the last *IGA* enables the validity flag in case no other assignment took place and implements the default reaction of the variable. Therefore, the default value of the variable should be already contained in the variable `x`. This is ensured by the *conclusion* that executes for all variables all delayed assignments. There, the default value must be assigned to all variables not written by a delayed assignment. This is possible, because the value of `x` is not used unless $\mathtt{x}_{\mathrm{v}}$ holds. Hence, the *conclusion* contains the following assignments related to the variable `x`:

$$
\bigwedge_{v \in \mathcal{V}} v_{\mathrm{v}} \Rightarrow \begin{bmatrix} \vdots & \\ x = & \begin{cases} v_1 & : \text{ if } \delta_1 \\ \vdots & \\ v_m & : \text{ if } \delta_m \\ \mathsf{defaultVal}(x) & : \text{ else} \end{cases} \\ x_{\mathrm{v}} = & \bigvee\limits_{i=1\ldots m} \delta_i \\ \vdots & \end{bmatrix}
$$

The default value is assigned by the else part of the case statement to ensure that all variables not written by a delayed assignment obtain this value. Hence, in the next step, it is enough to enable the validity flag once it could be decided that no other guarded action assigns a value to this variable and the default value must be used. This way we do not need an additional variable of the same type to store the value of the previous step.

Delayed Written Variables

One can see that either we set the validity flag $\mathtt{x}_{\mathrm{v}}$ for a variable `x` during a step, when an immediate assignment is executed or for the whole step, when a delayed assignment is executed.

The values for variables only written by delayed assignments, which is especially the case for all control-flow labels, can be determined before the next step starts: either one of the guards δ_i holds or the default reaction determines the value. Hence, the value contained in the variable is valid all the time and the corresponding validity flag is constantly true and therefore unnecessary. Hence, neither a carrier variable nor a validity flag is necessary for those variables.

Immediate Written Variables

Furthermore, the language Quartz differentiates two storage types: *event* and *memorized* variables. The default reaction of event variables resets the value to the default value of the variable's type. The default reaction of memorized variables is to transfer the previous step's value to the current step. This corresponds with the behavior of *IGAs* in case no assignment defines a variable during a transition. Hence, the explicit assignment of the default reaction could be omitted for *memorized* variables. Hence, only the assignment setting the validity flag to false is contained in the *conclusion*.

6.1.6 Examples

$$
\begin{array}{lcl}
\mathtt{mode} \land \mathtt{s2}_{\mathtt{V}} \land \neg \mathtt{r}_{\mathtt{V}} \land (\mathtt{l} \lor \mathbf{init}) & \Rightarrow & \begin{bmatrix} \mathtt{r} = \mathtt{s2} \\ \mathtt{r}_{\mathtt{V}} = \mathsf{true} \end{bmatrix} \\
\neg \mathtt{mode} \land \mathtt{m}_{\mathtt{V}} \land \neg \mathtt{r}_{\mathtt{V}} \land (\mathtt{l} \lor \mathbf{init}) & \Rightarrow & \begin{bmatrix} \mathtt{r} = \mathtt{m} \\ \mathtt{r}_{\mathtt{V}} = \mathsf{true} \end{bmatrix} \\
\neg \mathtt{r}_{\mathtt{V}} \land \neg \mathtt{l} \land \neg \mathbf{init} & \Rightarrow & \begin{bmatrix} \mathtt{r}_{\mathtt{V}} = \mathsf{true} \end{bmatrix} \\
\neg \mathtt{m}_{\mathtt{V}} \land \mathtt{mode} \land (\mathtt{l} \lor \mathbf{init}) & \Rightarrow & \begin{bmatrix} \mathtt{m} = (\mathtt{b} * \mathtt{c}) \\ \mathtt{m}_{\mathtt{V}} = \mathsf{true} \end{bmatrix} \\
\neg \mathtt{m}_{\mathtt{V}} \land \mathtt{s1}_{\mathtt{V}} \land \mathtt{s2}_{\mathtt{V}} \land \neg \mathtt{mode} \land (\mathtt{l} \lor \mathbf{init}) & \Rightarrow & \begin{bmatrix} \mathtt{m} = (\mathtt{s1} * \mathtt{s2}) \\ \mathtt{m}_{\mathtt{V}} = \mathsf{true} \end{bmatrix} \\
\neg \mathtt{m}_{\mathtt{V}} \land \neg \mathtt{l} \land \neg \mathbf{init} & \Rightarrow & \begin{bmatrix} \mathtt{m}_{\mathtt{V}} = \mathsf{true} \end{bmatrix} \\
\neg \mathtt{s2}_{\mathtt{V}} \land \mathtt{s1}_{\mathtt{V}} \land \mathtt{mode} \land (\mathtt{l} \lor \mathbf{init}) & \Rightarrow & \begin{bmatrix} \mathtt{s2} = (\mathtt{d} + \mathtt{s1}) \\ \mathtt{s2}_{\mathtt{V}} = \mathsf{true} \end{bmatrix} \\
\neg \mathtt{s2}_{\mathtt{V}} \land \neg \mathtt{mode} \land (\mathtt{l} \lor \mathbf{init}) & \Rightarrow & \begin{bmatrix} \mathtt{s2} = (\mathtt{c} + \mathtt{d}) \\ \mathtt{s2}_{\mathtt{V}} = \mathsf{true} \end{bmatrix} \\
\neg \mathtt{s2}_{\mathtt{V}} \land \neg \mathtt{l} \land \neg \mathbf{init} & \Rightarrow & \begin{bmatrix} \mathtt{s2}_{\mathtt{V}} = \mathsf{true} \end{bmatrix} \\
\neg \mathtt{s1}_{\mathtt{V}} \land \mathtt{m}_{\mathtt{V}} \land \mathtt{mode} \land (\mathtt{l} \lor \mathbf{init}) & \Rightarrow & \begin{bmatrix} \mathtt{s1} = (\mathtt{a} + \mathtt{m}) \\ \mathtt{s1}_{\mathtt{V}} = \mathsf{true} \end{bmatrix} \\
\neg \mathtt{s1}_{\mathtt{V}} \land \neg \mathtt{mode} \land (\mathtt{l} \lor \mathbf{init}) & \Rightarrow & \begin{bmatrix} \mathtt{s1} = (\mathtt{a} + \mathtt{b}) \\ \mathtt{s1}_{\mathtt{V}} = \mathsf{true} \end{bmatrix} \\
\neg \mathtt{s1}_{\mathtt{V}} \land \neg \mathtt{l} \land \neg \mathbf{init} & \Rightarrow & \begin{bmatrix} \mathtt{s1}_{\mathtt{V}} = \mathsf{true} \end{bmatrix} \\
\mathtt{m}_{\mathtt{V}} \land \mathtt{s2}_{\mathtt{V}} \land \mathtt{s1}_{\mathtt{V}} \land \mathtt{r}_{\mathtt{V}} & \Rightarrow & \begin{bmatrix} \mathbf{init} & = \mathsf{false} \\ \mathtt{l} & = \begin{cases} \mathsf{true} & \text{if } (\mathtt{l} \lor \mathbf{init}) \\ \mathsf{false} & \text{else} \end{cases} \\ \mathtt{m}_{\mathtt{V}} & = \mathsf{false} \\ \mathtt{s2}_{\mathtt{V}} & = \mathsf{false} \\ \mathtt{s1}_{\mathtt{V}} & = \mathsf{false} \\ \mathtt{r}_{\mathtt{V}} & = \mathsf{false} \\ \mathsf{mode} & = \mathsf{get}(\mathtt{mode}) \\ \mathsf{a} & = \mathsf{get}(\mathsf{a}) \\ \mathsf{b} & = \mathsf{get}(\mathsf{b}) \\ \mathsf{c} & = \mathsf{get}(\mathsf{c}) \\ \mathsf{d} & = \mathsf{get}(\mathsf{d}) \end{bmatrix}
\end{array}
$$

Fig. 6.9: Guarded Commands of the Example

In this section, we will illustrate the translation to *IGAs* by the running example and the examples presented in the introduction. All improvements of the previous section are already included in the *IGAs* shown here. The IGAs generated for Example 1 are:

$$\neg \mathtt{z}_v \wedge \mathtt{y}_v \Rightarrow \left\{ \begin{array}{l} \mathtt{z} = \mathtt{y} \\ \mathtt{z}_v = \mathbf{true} \end{array} \right\}$$
$$\neg \mathtt{y}_v \wedge \mathtt{x}_v \Rightarrow \left\{ \begin{array}{l} \mathtt{y} = \mathtt{x} \\ \mathtt{y}_v = \mathbf{true} \end{array} \right\}$$
$$\mathtt{y}_v \wedge \mathtt{z}_v \Rightarrow \left\{ \begin{array}{l} \mathtt{z}_v = \mathbf{false} \\ \mathtt{y}_v = \mathbf{false} \end{array} \right\}$$

Example 2 is represented by a single IGA – the conclusion, because it contains only delayed assignment:

$$\mathbf{true} \Rightarrow \left\{ \begin{array}{l} \mathtt{x} = \mathtt{y} \\ \mathtt{y} = \mathtt{x} \end{array} \right\}$$

The IGAs generated for Example 3 are:

$$\neg \mathtt{x}_v \wedge \mathtt{z}_v \Rightarrow \left\{ \begin{array}{l} \mathtt{x} = \mathtt{z} \\ \mathtt{x}_v = \mathbf{true} \end{array} \right\}$$
$$\neg \mathtt{y}_v \wedge \mathtt{z}_v \Rightarrow \left\{ \begin{array}{l} \mathtt{y} = \neg \mathtt{z} \\ \mathtt{y}_v = \mathbf{true} \end{array} \right\}$$
$$\mathtt{x}_v \wedge \mathtt{y}_v \wedge \mathtt{z}_v \Rightarrow \left\{ \begin{array}{l} \mathtt{z} = \neg \mathtt{z} \\ \mathtt{z}_v = \mathbf{true} \\ \mathtt{x}_v = \mathbf{false} \\ \mathtt{y}_v = \mathbf{false} \end{array} \right\}$$

Figure 6.9 contains the IGAs of the running example. The ten *SGAs* of Figure 6.7 are translated into thirteen *IGAs*. The value of each immediately written variable is determined by three *IGAs*: the translation of the *SGAs* (by transformation $\Gamma_{\mathcal{I}}$) generate two of them, and the third one represents the default reaction. The last *IGA* is the *conclusion*. The control-flow variables `run` and `l` are only written by delayed assignments, hence they do not need a valid flag and are only written in the *conclusion*. The valid flag of the other variables are explicitly set to false in the *conclusion*. The corresponding variables are not assigned a value, because the default reaction for memorized variables is to keep the value of the previous step. For the inputs, the *conclusion* contains an assignment of the currently read input value (get(i)).

6.1.7 Lifting Verification Results from *SGAs* to *IGAs*

The transformation Γ invalidates in general given temporal specifications, because one single macro step (of the synchronous module) now requires several intermediate steps. Hence, a temporal logic specification may no longer be satisfied by the introduced intermediate states. Therefore, this section explains how to use an existing method taken from a different application area to modify a given (valid) specification such that the *IGAs* execution still fulfills it.

In [GeMS13] and Chapter 5, transformations on temporal logic specifications were defined to lift available verification results for synchronous modules without abortion or suspension to refined temporal logic specifications that take care of the latter.

A suspension may postpone the current execution to a later point of time. The corresponding **suspend**-sensitive specification ensures that either the next macro step of the system is executed, and as a consequence, the specification must be satisfied for this step or the execution of the system is suspended, and a violation of the specification is negligible.

The transformation for the **suspend** case is given in Definition 34, were the used temporal operator $[\varphi \mathsf{W} \psi]$ represents the weak when operator [Schn03], which requires that φ must hold in the first state satisfying ψ. The crucial point of the definition is to forbid the use of a variable whenever the suspension takes place. Note again that all recursive calls will finally introduce a weak when operator and that the restrictions mentioned in [GeMS13] do not apply here.

For the approach, only the points of time where all variables have a valid value ($\mathsf{tick} := \bigwedge_{v \in \mathcal{V}} v_{\mathrm{v}}$) represent states where the specification should be evaluated. In Figure 6.10 this circumstance is presented by big states that are equivalent to the states of the synchronous execution and small states introduced by the execution of the *IGAs*. Hence using the condition $\neg\mathsf{tick}$ as 'suspend' condition σ, we are able to apply this transformation to any given specification of the synchronous module (like the two given in Figure 6.5).

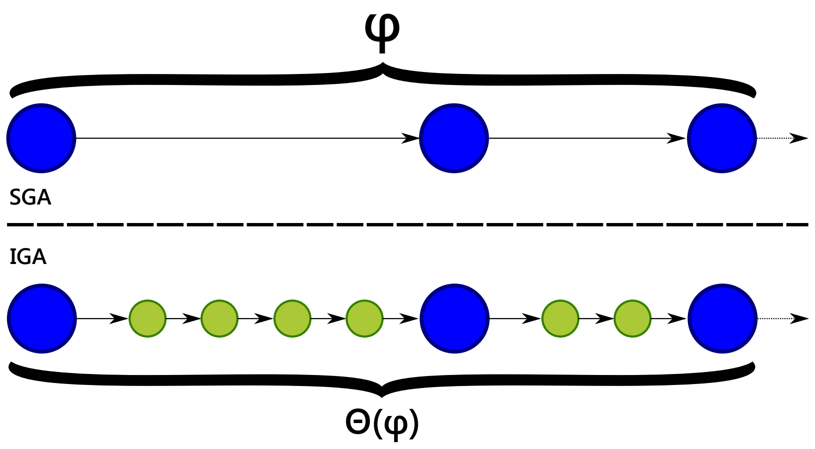


Fig. 6.10: States generated by IGAs compared to the states generated by SGAs

Furthermore, it is possible to strengthen the transformation given in Definition 34, because it allows an infinite suspension that is not wanted in our case. Hence, we require that only finitely many states are ignored in the evaluation, since a macro step must be evaluated in finitely many steps. Consider the following equivalent transformation based on the weak until operator instead of the weak when that helps to understand the required modification in that case:

$$\Theta^{\mathsf{st}}_{\mathsf{sp}}(\varphi,\sigma) := \begin{cases} [\sigma \mathsf{U} \varphi], & \text{if } \varphi \text{ is propositional} \\ \left[\sigma \mathsf{U}\, \mathbf{X}\Theta^{\mathsf{st}}_{\mathsf{sp}}(\psi,\sigma)\right], & \text{if } \varphi = \mathbf{X}\psi \\ \Theta^{\mathsf{st}}_{\mathsf{sp}}(\psi,\sigma) \otimes \Theta^{\mathsf{st}}_{\mathsf{sp}}(\gamma,\sigma), & \text{if } \begin{array}{l} \varphi = \psi \otimes \gamma \text{ with} \\ \otimes \in \{\wedge, \vee, \underline{\mathsf{U}}, \mathsf{U}\}. \end{array} \end{cases}$$

The evaluation of the specification φ is postponed by $[\sigma \mathsf{U} \varphi]$, hence to limit the suspension to a finite number of steps, it is enough to use the strong until operator ($[\sigma \underline{\mathsf{U}} \varphi]$) instead and replace the parameter σ by $\neg\mathsf{tick}$ to express that the specification must be evaluated only in states where tick holds.

Definition 41 (Transformation $\Theta(\varphi)$). *For a given temporal specification* $\mathsf{A}\varphi$*, the transformation* $\Theta(\varphi)$ *is defined as*

$$\Theta(\varphi) := \begin{cases} [\neg\mathsf{tick}\ \underline{\mathsf{U}}\ \varphi], & \text{if } \varphi \text{ is propositional} \\ [\neg\mathsf{tick}\ \underline{\mathsf{U}}\ \mathbf{X}\Theta(\psi)], & \text{if } \varphi = \mathbf{X}\psi \\ \Theta(\psi) \otimes \Theta(\gamma), & \text{if } \begin{array}{l} \varphi = \psi \otimes \gamma \text{ with} \\ \otimes \in \{\wedge, \vee, \underline{\mathsf{U}}, \mathsf{U}\}. \end{array} \end{cases}$$

6.1.8 Experimental Results

I implemented the presented transformation Γ along with the given transformations of Definition 34 and Definition 41. Then, I used SRI's Symbolic Analysis Laboratory (SAL)[1] that is a framework intended for performing abstraction, program analysis, and model checking to verify them.

All experiments were performed on an Intel Core 2 Duo with 2.66 Ghz (using Windows 7 and Cygwin). The table in Figure 6.11 contains the total execution time (given by SAL with option -v 1) in seconds for the running example with different N and transformed specifications:

- φ_{a1} : $\mathsf{G}\,[(\mathtt{mode} \rightarrow (\mathtt{r\ =\ a\ +\ b*c\ +\ d}))\ \mathsf{W}\ \neg\mathsf{tick}]$
- φ_{b1} : $\mathsf{G}\,[(\neg\mathtt{mode} \rightarrow (\mathtt{r\ =\ (a+b)*(c+d)}))\ \mathsf{W}\ \neg\mathsf{tick}]$
- φ_{a2} : $\mathsf{G}\,[\neg\mathsf{tick}\ \underline{\mathsf{U}}\ (\mathtt{mode} \rightarrow (\mathtt{r\ =\ a\ +\ b*c\ +d}))]$
- φ_{b2} : $\mathsf{G}\,[\neg\mathsf{tick}\ \underline{\mathsf{U}}\ (\neg\mathtt{mode} \rightarrow (\mathtt{r\ =\ (a+b)*(c+d)}))]$

The formulas in Figure 6.5 are used to generate all four specifications: The first two are determined by Definition 34 and the other two are obtained by Definition 41.

N	φ_{a1}	φ_{a2}	φ_{b1}	φ_{b2}	φ_{∞}
1	0.047	0.079	0.047	0.079	0.062
2	0.094	0.203	0.109	0.266	0.141
3	1.123	3.041	1.030	2.059	0.515
4	0.827	3.744	1.061	5.757	0.796
5	31.699	114.832	6.521	15.429	5.008
6	7.972	65.146	17.192	36.349	9.204
7	24.117	1061.212	24.321	75.895	17.035
8	9.688	13.478	9.157	21.995	10.404

Fig. 6.11: Execution Times (in sec) of SAL for the Running Example

The table shows that using the strict specifications φ_{a2} and φ_{b2} takes more time. Hence, it may be beneficial to use the weaker specification and to additionally prove that the condition `tick` occurs infinitely often by $\varphi_{\infty} = \mathtt{\textbf{G}\ \textbf{F}\ tick}$. The verification time for φ_{∞} is shown in the

[1] `http://sal.csl.sri.com/`

last column of the Table 6.11. The verification of the running example is faster splitting the specification, at least for the considered example. A further investigation on the scalability and the differences of both approaches is necessary.

Conclusion

A transformation from synchronous guarded actions (SGAs) to interleaved guarded actions (IGAs) was presented and implemented to connect the Averest system with tools based on IGAs like the symbolic analysis laboratory (SAL). Furthermore, an already existing transformation on specifications was adapted to transfer properties from the original SGA description to the new representation on IGAs. The approach was illustrated by verifying some properties of a running example with SAL. The approach presented in this section allows one to analyze, simulate, and verify synchronous programs, and especially Quartz programs, with well-established IGAs-based state-of-the-art tools that are designed for other MoCs. Future work is to extend this transformation for compound data types and to verify some larger examples by means of SAL. Since SAL already proved its capabilities in this area, the Averest system will also benefit from SAL. Another interesting idea is to use techniques presented in [BaBS12b] to relax the synchronous execution that was assumed here to an asynchronous one. This may lead to a smaller system description or to a more efficient verification.

6.2 Efficient Embedding of Synchronous Programs for Verification

In general, synchronous systems can be represented as a set of SGAs. While the synchronous semantics demands that *all enabled* actions have to be executed concurrently within the same variable environment, it is possible for certain sets of guarded actions to deviate from the synchronous execution scheme without changing the behavior. This is important to make use of tools like SRI's Symbolic Analysis Laboratory (SAL) that work with invariants and guarded actions, but only a subset of the enabled actions are chosen for execution. If the particular choice of the enabled guarded actions for execution is not determined, we may consider different choices that might influence the resource requirements needed for formal verification. In this section, we therefore investigate how three possible representations influence the runtime and memory requirements of automatic verification runs of SRI's SAL and answer the question "Whether changing the MoC allows one to verify synchronous systems faster".

This section and the corresponding publication [GeBS14] therefore explores the possibilities of representing a synchronous system in SRI's Symbolic Analysis Laboratory (SAL), and evaluate their effect on the performance of SAL's LTL model checker. To this end, we start in all cases with synchronous guarded actions as a general system representation. Since SAL only supports interleaved guarded actions, Section 6.1.3 and [GeSc13c] presented a possible translation of *SGAs* to SAL's *IGAs* to demonstrate that SAL can also be used to verify synchronous systems. This section aims at comparing that system representation against alternatives with respect to the performance of the later model checking. Therefore, the approach presented in [GeSc13c] (GC) will be compared with two others: One based on

SAL's synchronous composition of modules (SC) and another one based on a translation to equation systems (ES). Therefore, the tool *aif2sal* was implemented to generate these representations from an *SGAs*-based description of a Quartz program (see Figure 6.12).

The three transformations GC, SC, and ES are based on different paradigms that lead to different computations of a macro step in SAL. The ES representation describes the behavior of all *SGAs* in a single transition step by describing them as invariants. Therefore, no guarded action is required and an equation system must be solved in each reaction step. The SC transformation models the behavior of each variable by a single module containing all guarded actions writing this variable. The synchronous model of computation assures that only a single guarded action defines the value of a variable in a reaction step. Hence, in each reaction step all modules execute a single guarded action synchronously to define the behavior. This obliges SAL to resolve the data-dependencies between the modules. The GC approach describes the behavior in a single module by *IGAs*. This requires the explicit modeling of the data-dependencies as described in [GeSc13c]. Unlike the other approaches, the behavior of a single reaction step of the original system requires several transition steps in GC that requires the already described modification of the given specifications.

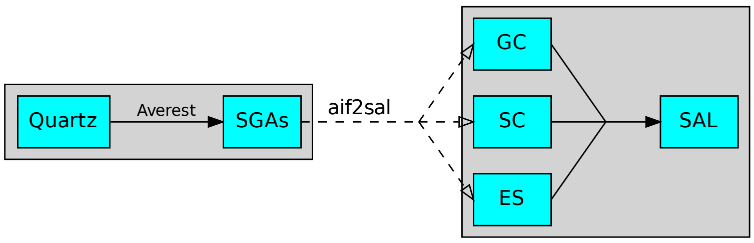


Fig. 6.12: Averest/SAL linkage

6.2.1 SRI's Symbolic Analysis Laboratory

SRI's Symbolic Analysis Laboratory (SAL)[2] is a framework intended for performing abstraction, program analysis and model checking, and it provides an intermediate language which will be the target of our translation process. A typical SAL system is represented by a context containing a set of modules and assertions. Each module declares a distinct set of inputs, outputs, local and global variables, as well as definitions (invariants) and transitions. Variables can be either current (X) or next variables (X'), where assignments to current variables take place in the current state, and to next variables in the following state. SAL allows to compose modules either synchronously (||) or asynchronously ([]). In synchronously composed modules, a transition from each module is executed simultaneously. With asynchronous composition however, an enabled transition from exactly one module is executed non-deterministically. Transitions can be written as an equation or as guarded commands. The equational format defines the trajectory of single variables, while the guarded

[2] http://sal.csl.sri.com/

commands define single transitions in the system. A guarded command of SAL contains, contrary to *SGAs*, a set of assignments and is enabled when its guard evaluates to true. Furthermore, SAL non-deterministically picks one of the enabled guarded commands and updates the next-state variables accordingly. The structure and behavior is equivalent to *IGAs* described in [GeSc13c]. A system without enabled guarded commands leads to a deadlock. A large set of tools such as symbolic/bounded model checkers, simulators and others, can then be used for analysis and verification.

6.2.2 Different Representations of Synchronous Systems

In this section, we present three different approaches of describing synchronous systems in SAL's input language. To ease the translation process, a tool called *aif2sal*, which is capable of converting AIF to the three different representations in SAL was developed.

Guarded Commands GC

Guarded commands in SAL are interpreted as interleaved guarded actions (*IGAs*), meaning that in each transition step an enabled guard action is non-deterministically chosen to define the step's behavior. This representation was already described in Section 6.1 and [GeSc13c] in detail.

Example

The SAL GC-representation of the running example has the structure shown in Figure 6.13. One can see that only a single valid flag (for the variable `o`) is required, because all other variables are written by delayed assignments. Additionally, the specifications were adapted to cover the changed temporal behavior. All newly introduced immediate states have in common that not all variables have a valid value and so the adapted specification only requires that the original specification is satisfied in states where all variables contain valid values.

Synchronous Composition SC

Another idea is to exploit SAL's synchronous composition primitive and divide the program into a set of synchronous modules. To that end, each variable, with the exception of inputs, will be represented as an independent module, and these modules will be then composed synchronously to provide the overall system behavior. It is worth noting that the semantics of the synchronous composition matches that of the synchronous model of computation. Since every variable has a unique value in each reaction step determined by a single *SGA* every module will execute exactly one transition.

In contrast to the GC transformation, the SC transformation is just a syntactic rewrite of the original program, in the sense that no guarded action will be modified. In this approach, data dependencies are resolved internally by SAL. The synchronous composition combines all definitions, initializations and transitions of the composed modules, taking care that the combination is still causally correct. In case of inconsistencies in the conjunction of the transitions are found, proof obligations are generated, but this problem does not apply here because the Averest compiler rules out causally incorrect programs.

```
ABROGC: MODULE =
BEGIN
  INPUT a,b,r : BOOLEAN
  OUTPUT o_v, o : BOOLEAN
  LOCAL init,wa,wb,wr,o_v : BOOLEAN
  INITIALIZATION [init = TRUE; wa = wb = wr = o = o_v = FALSE]
  TRANSITION [
    [] ¬o_v ∧¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa) ⟶
        o_v' = TRUE ;
        o' =  TRUE ;
    [] ¬o_v∧(r∨¬(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa)) ⟶
        o_v' = TRUE ;
    [] o_v ⟶
        wr' =  ¬r∧(wr∨(a∧wa∧b∧wb)∨(b∧wb∧¬wa)∨(a∧wa∧¬wb));
        wb' =  ¬r∧wb∧¬b∨r∨init;
        wa' =  ¬r∧wa∧¬a∨r∨init;
        init' =  FALSE ;
        o' =  FALSE ;
        o_v' = FALSE ;
    ]
END;
s1 : THEOREM ABROGC ⊢ AG[¬o_v U (o ⇒ a ∨ b)];
s2 : THEOREM ABROGC ⊢ AG[¬o_v U (o ⇒ X¬o)];
```

Fig. 6.13: GC Representation

Example

The translation of the ABRO example to SC consists of separating *SGAs* by their written variable into individual modules as depicted in Figure 6.14a (for variable o). Each module will have every other variable that is read by the *SGAs* as input and a single output being the writable variable itself. All *SGAs* for each variable are then collected, and used to properly initialize the module and to describe its transitions as guarded commands.

```
oMod : MODULE =
BEGIN
  INPUT  a, b, r, wa, wb : BOOLEAN
  OUTPUT o : BOOLEAN
  INITIALIZATION
    [o=¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa)]
  TRANSITION
    [  ¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa)
            ⟶ o' = TRUE;
    [] ELSE ⟶ o' = FALSE; ]
END
```

(a) SC: Single module

```
 wOMod  : MODULE = ...
 waMod  : MODULE = ...
 wbMod  : MODULE = ...
 wrMod  : MODULE = ...
  oMod  : MODULE = ...
ABROSC  : MODULE = initMod
                   || waMod
                   || wbMod
                   || wrMod
                   ||  oMod;
s1 : THEOREM ABROSC ⊢ AG(o ⇒ a ∨ b);
s2 : THEOREM ABROSC ⊢ AG(o ⇒ X¬o);
```

(b) SC: Composition

Fig. 6.14: Single Module and Synchronous Composition

In Figure 6.14b, the synchronous composition of the necessary main module is shown. We simply compose all writable variables (**`init`**, `wa`, `wb`, `wr` and `o`) into a single module. It is important to note that a composed module will lead to a deadlock whenever at least one of the modules is in a deadlock, hence we introduce an **ELSE** guard to guarantee that there is always a transition to be taken. Moreover, the **ELSE** guard will assign the default value to the variable, which is effectively the default reaction.

Equation System ES

This transformation converts the *SGAs* into equations (one per variable). It is usually not trivial to generate such an equation system, and a corresponding transformation is already implemented in the Averest system (see Section 2.3.7). The translation of *SGAs* to equations will generate exactly one equation for each output, local and location variable. Furthermore, an additional carrier variable must be added for each variable to which an immediate and delayed assignment is made. The carriers will simply hold the value until the next point of time.

The execution of an equation system under the synchronous model of computation is then as follows: in every macro step new input variables are read and all of the equations are evaluated with regards to the newly read values. The resulting right-hand side of each equation is then assigned to its respective variable.

This can be easily done in SAL by using definitions instead of guarded commands. Definitions in SAL are of the form $\langle X = \texttt{EXPR}\rangle$ for the current state or $\langle X' = \texttt{EXPR}\rangle$ for the next state. In contrast to guarded commands, which are picked individually, all definitions are evaluated in every state and the resulting value for the expression EXPR is assigned to the variable. Once we have the *SGAs* given as equations, the translation to SAL is

```
ABROES : MODULE =
BEGIN
  INPUT a, b, r : BOOLEAN
  OUTPUT o : BOOLEAN
  LOCAL wa, wb, wr, w0 : BOOLEAN
  INITIALIZATION [init = TRUE; wa = wb = wr = FALSE]
  DEFINITION o = ¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa);
  TRANSITION
    init' = FALSE;
    wa' = ¬r∧wa∧¬a∨r∨init;
    wb' = ¬r∧wb∧¬b∨r∨init;
    wr' = ¬r∧(wr∨(a∧wa∧b∧wb)∨(b∧wb∧¬wa)∨(a∧wa∧¬wb));
END;
s1 : THEOREM ABROES ⊢ AG(o ⇒ a ∨ b);
s2 : THEOREM ABROES ⊢ AG(o ⇒ X¬o);
```

Fig. 6.15: ES: Single module

straightforward. We will have a single module containing the original inputs and outputs and all other variables as local variables, as seen on Figure 6.15 for the ABRO example. SAL supports in the **DEFINITION** section only assignments to variables of the current state, hence a distinction between **INITIALIZATION**/**TRANSITION** and **DEFINITION** is necessary. All equations defining a next-state variable must be initialized in the **INITIALIZATION** section and described in the **TRANSITION** section. Hence, the immediate assignments will be represented as an invariant in the **DEFINITION** section and will be evaluated in every state, including the initial one. The **INITIALIZATION** section will be evaluated in the initial state and contains the initialization of variables written by delayed assignments like the location variables and the carriers[3] to their default values. The **TRANSITION** section contains equations for evaluating the next-state.

Note that problems like the default reaction were already handled during the translation to an equation system by simply adding an extra branch to every equation, assigning the variable's default value whenever none of the previous conditions hold.

6.2.3 Experimental Results

The presented transformations were used to verify and benchmark the experiments using SAL's symbolic model checker (sal-smc). All experiments[4] were performed on a Intel® Core™ i5-3470 CPU @ 3.20GHz using Ubuntu 13.04.

In the following, we briefly describe each example, in terms of what they do, number of variables in the original Quartz program, number of *SGAs* after compilation, and the number and kind of properties that were verified for each. This gives an idea on the complexity of each example:

P	#SGA	#GC	#SC	#ES	*GC*	*SC*	*ES*
ABRO	7	4	6(12)	5	0.11	0.06	0.05
ABROM[M=10]	23	2	14(36)	13	0.74	1.13	0.50
ABROM[M=13]	29	2	17(45)	16	4.27	7.92	3.27
AuntAgatha	2	4	3(4)	3	0.12	0.07	0.09
VendingMachine	23	23	12(32)	11	1.14	0.15	0.07
LightControl	36	25	12(47)	11	1.79	0.44	0.40
MinePumpController	42	41	22(61)	21	7.60	0.22	0.09
RSFlipFlop	7	2	5(11)	8	53.51	1.18	1.18
MemoryController	41	28	17(87)	31	407.95	42.93	3.42
IslandTrafficControl	83	62	35(109)	36	504.64	62.40	1.94
FischerMutex	60	15	16(121)	25	0.14	0.22	0.09
Dekker	28	14	14(61)	15	0.63	0.21	0.17
SingleRowNIM	15	14	6(31)	8	0.06	0.04	0.04
PigeonHole	1	7	2(5)	31	0.01	0.05	0.05
Queens	1	7	2(5)	37	0.29	0.19	0.20
MagicSquare	29	35	4(65)	11	1.83	65.67	9638.84
Search_OlogN	13	17	7(33)	11	97.19	136.18	47.21

Fig. 6.16: Size of the Representations and Execution Times (in sec) of SAL

[3] Carriers are present only when the program utilizes immediate and delayed assignments for a single variable, which is not the case for ABRO.

[4] All examples a publicly available under `http://www.Averest.org/examples`.

ABROM is a larger version of the ABRO example which waits for M events in parallel instead of just two. It contains **22** *SGAs* for $M = 10$ or **28** for $M = 13$, **2** inputs, **1** output and **3** safety properties.

AuntAgatha is an implementation of an old puzzle where the reader has to find who killed Aunt Agatha in Dreadsburry Mansion based on simple boolean statements. The problem is represented by **2** *SGAs*, **21** inputs, **1** output and **3** boolean properties.

VendingMachine is a vending machine controller that dispenses gum in reaction to the insertion of nickels and dimes, which is described by **2** *SGAs*, **2** inputs, **2** outputs, and **3** safety properties.

LightControl models the light control system of a room with regards to to its occupancy. Its functions include switching the light on/off, dimmer control and notification of alarms. The implementation contains **36** *SGAs*, **22** inputs, **12** outputs, and **10** safety specifications.

MinePumpController starts or stops the pump of a mine according to alerts issued by the carbon dioxide and methane monitors, as well as the water level. It contains **40** *SGAs*, **27** inputs, **30** outputs, and **7** safety specifications.

RSFlipFlop describes a RS-Flipflop with NOR-gates of equal delay, modeled as a single macro step. It contains **7** *SGAs*, **2** inputs, **2** outputs, and **8** specifications (three safety and five co-Büchi).

MemoryController models a memory controller providing mutual exclusion by maintaining region locks for addresses. The implementation contains **41** *SGAs*, **5** inputs, **12** outputs, and **8** safety specifications.

IslandTrafficControl: An island is connected via a tunnel with the mainland. Inside the tunnel is a single lane so that cars can either travel from the mainland to the island or vice versa, which is signaled by traffic lights on both ends of the tunnel. It is represented by **75** *SGAs*, **15** inputs, **32** outputs, and **13** specifications (eleven safety and two Büchi) modeled in **5** modules.

FischerMutex implements Fischer's protocol which is used to control mutual exclusive access of processes to a shared ressource. It contains **2** inputs and **3** outputs used by **60** *SGAs* and two specifications.

Dekker: Dekker's mutual exclusion algorithm with **2** inputs and **two** outputs used by **28** *SGAs*and two specifications.

SingleRowNIM: Single Row NIM Game with Strategic Player A. The implementation contains **15** *SGAs*, 4 inouts and a single specification.

PigeonHole described the problem to put several pigeons in a number of holes [GrZa03]. The implementation contains **1** *SGAs*, a matrix as inout and a single specification.

Queens describes the problem of placing N Queens on a NxN chessboard so that no queen can attack another queen. A single *SGAs*, a matrix as inout and a single specification are contained in the implementation.

MagicSquare considers a number of magic squares, such that in each square the numbers from 1 to 9 should be placed such that the sums of all rows and columns is the same. The implementation consist of **29** *SGAs*, **2** outputs, a matrix as inout and a single specification.

Search_OlogN: This is the well-known binary search algorithm that expects a sorted array and a value whose membership in the array is to be checked. **13** *SGAs* are required to represent this algorithm. The contained specification ensures the termination of the algorithm.

The table in Figure 6.16 roughly measures the size of the original program regarding the number of *SGAs* (#SGA), as well as the size of each representation in terms of the number of guarded commands (#GC) for GC, number of modules and guarded commands (#SC[5]) for SC, and the number of equations (#ES) for ES. Interestingly, ABROM contains only 2 guarded commands in GC, while having about 20 *SGAs* in the original Quartz program.

[5] The number inside parenthesis is the sum of the number of guarded commands in the context.

This happens because ABROM only features delayed assignments, which according to the transformation described in 6.2.2, are combined into a single guarded command (the *conclusion*). This also explains the particularly good performance for GC on verifying it (Figure 6.16). The number of equations used in ES corresponds with the number of modules used for SC. For each variable, an equation is contained in ES and SC contains besides the module for the synchronous composition for each variable a module. In case a carrier variable has to be introduced for ES, they will differ.

P	GC		SC		ES	
	#V	#N	#V	#N	#V	#N
ABRO	34	786	30	616	14	183
ABROM[M=10]	80	2289	98	3855	52	1221
ABROM[M=13]	98	3339	122	3381	64	2065
AuntAgatha	100	1434	40	130	48	180
VendingMachine	130	10855	138	4496	22	384
LightControl	98	7108	114	6908	44	837
MinePumpController	126	98949	136	8500	36	636
RSFlipFlop	36	1883	38	929	24	1010
MemoryController	158	931422	188	92815	98	48134
IslandTrafficControl	144	773001	212	58217	58	30401
FischerMutex	2378	88	122	4851	64	1735
Dekker	61	8437	86	4342	36	5317
SingleRowNIM	48	1148	46	637	24	310
PigeonHole	132	650	66	254	64	249
Queens	156	874	78	312	76	301
MagicSquare	116	15524	98	553	76	3699679
Search_OlogN	270	9315	162	33098	138	16718

Fig. 6.17: BDD size in terms of the number of variables (#V) and number of nodes (#N)

As for the actual performance of each representation, Figure 6.16 shows that ES is in almost every benchmark faster than GC or SC. In the worst case (IslandTrafficControl), it was more than 250 times faster than an equivalent program in the GC representation and roughly 32 times faster than SC. Interestingly is the fact that for the MagicSquare example the opposite is the case, there the GC representation is roughly 35 times faster than SC and more than 5000 times faster than ES. Not surprisingly, the complexity of the properties can also increase the verification time in certain cases, as with the GC representation for RSFlipFlop, which regardless of being a fairly minimal Quartz program, contains complex assertions concerning the stability of the circuit. To further corroborate that the ES representation is indeed the best representation, we measured the size of the BDD with respect to the number of variables (#V) and the number of nodes (#N) for each representation. As seen on Figure 6.17, the size of the BDD for the ES transformation was usually smaller than its counterparts, which certainly relates strongly to the times measured in Figure 6.16.

6.2.4 Conclusions

We considered three different ways to represent synchronous systems in SAL's transition language, and evaluated them concerning the performance of model checking. The chosen representations differ in the number of guarded actions that are chosen for execution in every reaction step. As a result, we can clearly say that the ES transformation, which represents the system as definitions (equations) is in general the best choice (for SAL).

Hence, (at least for SAL) a transformation from the synchronous MoC to the asynchronous MoC SAL is based on does not result in a better verification task in terms of time or memory usage. The runtime and the memory usage of the SC and ES representations, which are more or less equivalent with the synchronous MoC are better.

Chapter 7

Evaluation

In this chapter, the implementation and the main data structures of the AIFProver is described. Due to its specialized application domain, the tool does not aim to replace state-of-the-art theorem provers. Instead, a specialized and convenient interactive theorem prover for the verification of synchronous systems is developed. Moreover, since the presented verification tool is tightly integrated with a tool for hardware and software synthesis, many software libraries can be shared between synthesis and verification. In the long term, synthesis can therefore also benefit from verification in that optimizations are possible that are not used today due to too pessimistic estimations.

As outlined in [GeSc12] and Chapter 3, difficult problems arise when source code descriptions should be directly used for interactive verification of synchronous programs. For this reason, in [GeSc12a] and Chapter 4, it was suggested to work on the intermediate representation of synchronous guarded actions that allows one a more flexible proof goal decomposition.

The idea of the approach was already presented in Chapter 4 the rules determined by the user are applied by the AIFProver as shown in Figure 7.1.

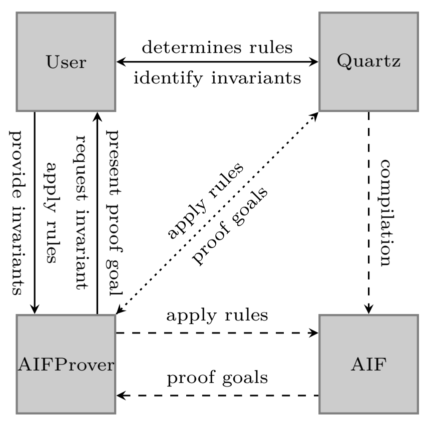


Fig. 7.1: Idea of our Approach

7.1 Implementation Details of the AIFProver

Averest and the AIFProver share data structures for types, expressions, specifications, and guarded actions as well as available transformations on sets of guarded actions like reduction to Boolean types (which allows the use of SAT solvers). This does not only avoid reimplementations, it also immediately assures that the prover and the synthesis framework will always remain consistent in case the language is extended or changed. For this reason, one has to implement proof goals, theorems and proof rules in the AIFProver. Since Averest has been implemented in Microsoft's new programming language F#, the AIFProver is also implemented in F#, but due to the capabilities of the .NET framework, any other programming language like C# could have been used as well. The choice of F# was however also made to directly use F# sessions for interactive verification following the spirit of LCF style theorem provers. It is the second version of the AIFProver, which is described here. The previous version was based on the results of [GeSc12a] that focused on the verification of assumption and assertions.

7.1.1 Proof Goals

A proof goal has an identifier `id` to address different proof goals. Moreover, it contains system defined (`sasm`) and user defined (`uasm`) assumptions that can be used for the proof. The EFSM state is encoded by the set of names of the control-flow locations (`labels`) that hold in that state. Moreover, the set of delayed assignments (`prevStep`) that were executed in the previous step (and will therefore affect the current state) is stored. The core of the proof goal is an AIF system itself! Note that guarded actions do not only consist of assignments, but also of assumptions, assertions and temporal logic specifications. Thus, the F# type for a proof goal is defined as follows:

```
type ASM = QName * SpecExpr
type proofGoal = {
  id       : string
  sasm     : ASM list
  uasm     : ASM list
  labels   : Set<QName>
  prevStep : GrdAction list
  system   : AIFSystem
}
```

The initial step and all other steps of a program has to be distinguish to deal with the semantics of past temporal operators and the initialization of variables. Hence, the following discriminated union is used to implement it:

```
type ProofGoal = InitGoal  of proofGoal
               | GenGoal   of proofGoal
```

This allows us to restrict the application of rules for the initial state to `InitGoals` and all other rules to `GenGoals`.

7.1.2 Theorems

Proved tasks will be stored as theorems, with the type *Thm* that consists of a similar discriminated union:

```
type Thm = InitThm  of proofGoal
         | GenThm   of proofGoal
```

The difference of the two kinds of theorems is that `InitThm` are allowed to use only for `InitGoals`.

7.1.3 AIF Proof

An entire proof is represented by the data type `aifproof`. It contains an unmodified copy of the originally considered AIF file (`proofSystem`), a list of all still unproved (`ProofGoals`) and proved (`provedTHM`) sub-goals as well as a global list of assumptions (`proofASM`).

```
type aifproof = {
        proofSystem : AIFSystem;
        ProofGoals  : (ProofGoal * ProofGoal list) list;
        proofASM    : ASM list;
        provedTHM   : Thm list
       }
```

7.1.4 Proof Rules

Proof rules are therefore F# functions that map existing theorems to new theorems. Also, we implement *tactics* which decompose desired proof goals into a list of subgoals. Starting with a proof goal, a proof tree is generated by applying a tactic to a leaf of the current proof tree. In our implementation, only the leafs of the current proof tree are stored, and if all leafs were finally proven by a decision procedure, the original root becomes a theorem (i.e. a proven proof goal). Proof rules and tactics correspond to each other and are used in forward proofs (where one derives theorems from axioms by rules) and backward proofs (where proof goals are subsequently decomposed into trivial sub-goals), respectively.

The rules listed in Chapter 4 are the basis of the AIFProver and operate at the level of macro steps and have to introduce assumptions and assertions when a proof goal is decomposed into subgoals in order to implement an *assume-guarantee* deduction system [HeQR00]. For example, consider Figure 7.2: It illustrates the introduction of assumptions and assertions when splitting a sequence $P_1; P_2$ into sub-goals. To this end, the corresponding proof rule is given an intermediate specification φ that is added by the rule application as assertion for P_1 and as assumption for P_2. For this reason, there are assumptions and assertions that are added by rule applications and others that were part of the original proof goal.

Finally, rules are implemented by the following F# type:

```
type Rule   = Thm list -> Thm
```

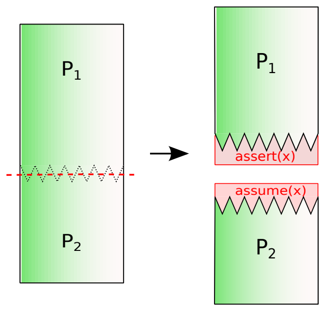


Fig. 7.2: Decomposition of Sequence $P_1;P_2$

7.1.5 Proof Management Rules

Besides the described rules for decomposing or proving a (sub-)goal, an interactive verification tool needs to apply the implemented rules, print or skip the current proof goal and rules that make the usability of the system convenient. Additionally, rules for the introduction of new lemmata or the usage of existing lemmata are required.

7.1.6 Tactics

Tactics are a bunch of compiled rules that are used often together. One of the most important tactics is *AutoTac*, which applies the rules to rewrite immediate assignments, replaces case statements, splits conjunction statements in the conclusion to separate proof goals, shifts implications in the conclusion to the assumptions and tries to prove proof goals before disjunctions are unstitched and a second solver run is made. Each rule is applied to all new generated proof goals until no changes occur and then the next rule is used.

Besides other tactics there are simplification tactics and a tactic that decomposes a proof goal into several goals, depending on a case-statement, which is the usual representation of an equation.

Finally, tactics are implemented by the following F# type:

```
type Tactic = ProofGoal list -> ProofGoal list
```

7.1.7 Structure of the AIFProver

Figure 7.3 shows the structure of the AIFProver and the embedding in the Averest system. First, a Quartz program is compiled to an AIF file and fed into the AIFProver. The user decides then which rules are applied by inspecting the source code or simulation results coming from the simulator contained in the Averest system. These rules prove a goal or decompose them into several sub-goals.

7.2 Verifying the ABRO Example

In this section, a proof for the `ABRO` module shown in Chapter 2 is presented. The proof is started by calling the AIFProver with the AIF file's path as parameter. The AIFProver creates

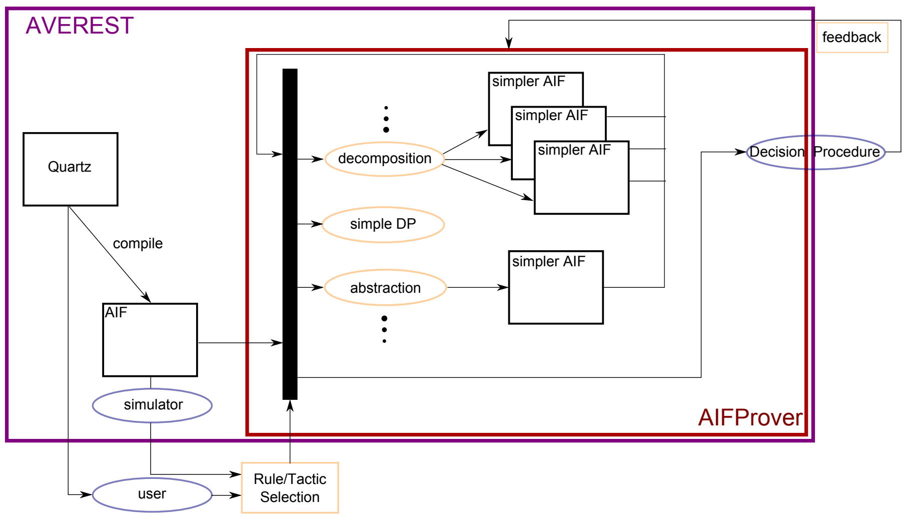


Fig. 7.3: Structure of the AIFProver

a initial proof template for the given AIF in case no proof script is found (see Figure 7.4) that loads the required libraries and defines each variable contained in the AIF file as variable of

```
open System
#r "Averest.Core.dll"
open Averest.Core
open AIF
open Types

#r "Averest.Compilation.dll"
open Averest.Compilation
open Statements
...

Console.Out.WriteLine (" ");
Console.Out.WriteLine ("==================================================");
Console.Out.WriteLine ("  Welcome to the interactive Averest environment");
Console.Out.WriteLine ("  (Version is {0})", AverestVersion);
Console.Out.WriteLine ("==================================================");
Console.Out.WriteLine ("");
Console.Out.WriteLine ("");
Console.Out.WriteLine ("");
ShowCorrectness (fsi.CommandLineArgs.[0]);;

//Defining QNames as Variables...
let a = GetQName "a";;
let b = GetQName "b";;
let r = GetQName "r";;
let o = GetQName "o";;
let bootflag = ____running003
let wa = GetQName "wa";;
let wb = GetQName "wb";;
let wr = GetQName "wr";;
```

Fig. 7.4: Prooftemplate for ABRO

the AIFProver for a convenient use. The AIFProver then starts a F# interactive session and loads the corresponding proof script. The proof of an AIF file is initiated by the function ShowCorrectness(path) (contained in the proof template). This function translates the AIF file to a proof goal and generates for each specification a proof task. The Figures 7.5, 7.7, and 7.8 contain the important part of the F# script to prove the ABRO example (all lines that are part of the initial proof script are omitted).

```
ShowForall();;
RWSWIAs();;
Auto();;
AssertThm("s1");;
```

Fig. 7.5: Proof for Specification s1

The first proof task is simple and is proven by rewriting the specification with the definitions of the variables and using one of the most important tactics called `AutoTac`, which rewrite with immediate assignments, replaces case statements, splits conjunctions in the conclusion, shifts implications in the conclusion to the assumptions and tries to prove *ProofGoals* before case splits are performed (see Figure 7.5). The last function call checks if the specification is successfully proven and raises an error otherwise. This function allows one to check if a proof still works after some changes in the proof script or the proof rules are performed.

```
A G o → X !o
 ⇒ (!r&(a&wa&b&wb|!wa&b&wb|!wb&a&wa) ⊨
   !(!X(r)&(X(a)&X(wa)&X(b)&X(wb)|!X(wa)&X(b)&X(wb)|!X(wb)&X(a)&X(wa))
 ⇒ (X(wa) = ... & X(wb) = ... & init = !(wa|wb|wr) & !r&(a&wa&b&wb|!wa&b&wb|!wb&a&wa)
      ⊨ !(!X(r)&(X(a)&X(wa)&X(b)&X(wb)|!X(wa)&X(b)&X(wb)|!X(wb)&X(a)&X(wa))
 ⇒ (X(wa) = ... & X(wb) = ... & !r&(a&wa&b&wb|!wa&b&wb|!wb&a&wa)
      ⊨ !(!X(r)&(X(a)&X(wa)&X(b)&X(wb)|!X(wa)&X(b)&X(wb)|!X(wb)&X(a)&X(wa))
```

Fig. 7.6: Intermediate Result for Specification `s2`

The proof of specification `s2` given in Figure 7.7 uses the definition of variable `o` to rewrite the specification (`RWSWIA`). After this, the definitions of the variables `wa` and `wb` are inserted in the assumptions. Then, the fact that the boot flag **`init`** holds if no other label holds is used. The intermediate proof state is given in Figure 7.6. Finally, tactic `RW_X` is used to shift the **`next`** operator inwards to variables so that the obtained goal is proved by `AutoTac`.

Alternatively, one may define the function SolveCase (again Figure 7.7) and apply the rule to split a disjunction together with the new defined function several times.

In the beginning of the proof of specification `s3`, the proof goal is simplified and decomposed into several sub-goals, where all but two are provable by Solve_Tac. One of the remaining two proof goals state that the control flow of ABRO rests in `wa` but not in `wb` and it must be shown that the specification **`PWX`** `[b` **`PWB`** `r]` is fulfilled. The other proof goal states the same with `wa` and `wb` swapped. Since, leaving `wb` is only possible by an occurrence of `b`

```
ShowForall ();;
RWSWIA o;;
UseVarDef wa;;
UseVarDef wb;;
UseRunningDef bootflag
RWSWADs();;
RWCases();;
RWImplSpec();;
RWImplSpec();;

////Alternative:
//RW_X()
//Auto()

let SolveCase () =
    RW_X()
    let _ = RWImplSpec()
    RWAsm()
    RWSpec()
    let _ = RWImplSpec()
    RWAsm()
    RWSpec()
    Solve()

SplitDisj();;
SolveCase();;
SplitDisj();;
SolveCase();;
SolveCase();;

AssertThm ("s2");;
```

Fig. 7.7: Proof for Specification s2

after the last occurrence of `r`, the specification **PWX** `[b` **PWB** `r]` is fulfilled. Two lemmata are required to prove these proof goals. These lemmata are introduced by the function NewLemma, which parses the second parameter as a specification and generates a new proof task with this specification in the context of the current AIF. Both lemmata state analogously to the remaining proof goals that for all points of time where the control flow rests in `wa` but not in `wb` (and `wb` but not in `wa` respectively) the specification **PWX** `[b` **PWB** `r]` (and **PWX** `[a` **PWB** `r]` respectively) are fulfilled. This is the case, because the control flow leaves label `wa` and `wb` respectively only iff `a` and `b` respectively is satisfied after the last occurrence of `r`, hence the specification **PWX** `[b` **PWB** `r]` (and **PWX** `[a` **PWB** `r]` respectively) is fulfilled. This is shown by induction using F# functions as macro to apply the same rules to the

```
ShowForall()
RWSWIAs();;
RWSWADs();;
RWConj();;
Applys Next_PWBTac;;
Applys Strip_Tac;;
Applys Solve_Tac;;

NewLemma "Lemma1" "A G(wa&!wb -> PWX[b PWB r])";;
NewLemma "Lemma2" "A G(!wa&wb -> PWX[a PWB r])";;

let checkLemma12 () =
    let checkBase () =
        Induction()
        Chain [Gen_To_InitTac;  RwswiasTac; Solve_Tac]
    let checkStep () =
        let _ = Applys RWImplTac
        let _ = Chain [RW_PWXTac; RwswnvdTac wa; RwswnvdTac wb;
                       RWCasesTac; RwswadsTac; RWAsmTac]
        let _ = Chain [Cases_Tac bootflag; Cases_Tac r]
        let _ = Applys Solve_Tac
        Chain [Next_PWBTac; RWConjTac; Solve_Tac; DisjL2ImplTac;
               RWImplTac; RWAsmTac; RWSpecTac; MPTac; Solve_Tac]
    let _ = checkBase()
    checkStep()

checkLemma12();;
AssertThm("Lemma2");;
checkLemma12();;
AssertThm("Lemma1");;

UseThm "Lemma2";;
SpecializeAsmContains "PWX";
RWImplSpec();;
MP();;
Solve();;

UseThm "Lemma1";;
SpecializeAsmContains "PWX";
RWImplSpec();;
MP();;
Solve();;

AssertThm("s3");;

PrnThms();;
```

Fig. 7.8: Proof for Specification s3

different proof tasks. Additionally, another F# function is used to prove the remaining proof goals of specification `s3`.

7.3 Extending the Implementation for Modular Verification

After the short description of the AIFProver that was implemented following the rules of Chapter 4, this section describes the necessary extensions to enable modular verification as presented in Chapter 5.

7.3.1 Proof Goals

For the modular verification approach one has to distinguish between proof goals that consider closed systems, and others where an arbitrary environment may be added (open systems). The latter is required for a modular verification and differs from closed system verification in that the reaction to absence is left out, and instead variables may have arbitrary values in these cases. Hence, the type for proof goals is extended:

```
type ProofGoal = InitClosedGoal  of proofGoal
               | GenClosedGoal   of proofGoal
               | InitModularGoal of proofGoal
               | GenModularGoal  of proofGoal
```

7.3.2 Theorems

Proved tasks will be stored as theorems with the type *Thm* that consists of a similar discriminated union:

```
type Thm = InitClosedThm  of proofGoal
         | GenClosedThm   of proofGoal
         | InitModularThm of proofGoal
         | GenModularThm  of proofGoal
```

The only difference in the rules that must be made is that the default reaction is not used iff the proof goal is an `InitModularGoal` or a `GenModularGoal`. The use of `InitModularThm` or a `GenModularThm` in the proof of closed system is possible without problems.

Conclusions

This chapter described the implementation of an interactive tool for the verification of synchronous systems that is tightly integrated with the Averest synthesis framework. This allows an efficient reuse of existing code and a lightweight implementation of proof rules so that experiments with different sets of proof rules are easily possible. After a preliminary implementation based on the rules presented in [GeSc12a] and the use of these rules to verify some benchmark examples the AIFProver presented in this chapter was implemented. Proofs can be stored as F# scripts so that one can try to rerun the proofs if minor changes were made in the systems or specifications. The AIFProver represents now a basic implementation for the verification of synchronous systems. Next steps could be the extension of the presented proof rules to allow proofs of industrial-sized systems or to use techniques to increase the reliability, e.g. certify AIFProver proof by another verification tool or the embedding into a theorem prover.

Chapter 8

Conclusions

This work represents a feasibility study to enable interactive verification for synchronous systems. The aim was to identify a representation of proof goals and a corresponding set of rules operating on them to verify synchronous systems in an interactive manner.

Therefore, the definition of Hoare-Calculus-like rules on the source-code level were discussed and the problems to define these rules were presented. One of the main issues was that in a Quartz program several control-flow locations may be active and so the compositional approach of Hoare is difficult to implement.

To circumvent most of the encountered issues and to reduce the number of required rules, a normal form for the source code was proposed that aggregates all micro step assignments into a single synchronous tuple assignment (STA) such that the reasoning about the behavior was simplified by collecting the program's behavior at a single program part (the STA). This normal form does not contain any parallel statement and therefore allows again to follow the compositional approach defined by Hoare. This way, a convenient use of interactive verification based on an adapted Hoare calculus was possible. Unfortunately, the required program transformation at the source-code level was proven to be impossible in general without adding additional variables (see Theorem 1), which complicates the verification task for the user and was the reason to alter the approach again. Nevertheless, two incomplete transformations covering most Quartz programs were presented.

Afterwards, a complete approach to define interactive verification for synchronous systems was presented that defined rules for the compiled Quartz code and circumvented the identified problems for the source-code approaches and did not rely on the SSTA form. This approach has several advantages: the techniques and methods implemented in the compiler are reusable without additional effort; using guarded actions instead of the original source code allows a more flexible decomposition of proof goals – there is no need to follow the syntax of the program. The user is able to determine the rules to apply from the source-code, as a result of the strong relation between source-code and guarded actions through the label variables. Furthermore, this approach was extended to handle LTL specifications and special rules for module calls and pre-emption statements were added. This allowed the verification of synchronous systems in an interactive manner and leads to the implementation of the AIFProver prototype.

The goal of the thesis was reached by the implementation of the AIFProver that enabled interactive verification of synchronous systems.

This work allows one to start now with the development of a tool to allow proofs of industrial-sized systems. Therefore, the presented approach needs to be embedded into a theorem prover or other tools to improve the reliability and usability. This was not possible before, because the kind of representation and the domain of the rule set was not known. Directly starting with the embedding could have resulted in an huge overhead, because the basis of the final approach was changed so often during this work such that this would have supersede the required proofs for the embedding.

An improvement for the AIFProver would be a deeper integration of the existing decision procedures, e.g. model checking, such that generated counterexamples are somehow usable to ease the verification task or identify where the user should start again with his/her proof. Additionally, the set of rules might be extended by demand.

An extension of the AIFProver for the recently established modeling of cyber-physical systems by HybridQuartz [Baue12] to verify these systems is another possible further work. The verification of cyber-physical systems in literature is immature and the determinism of HybridQuartz allows to adapt verification techniques, but the state space will be enormous, which requires similar techniques used in this thesis.

Additionally, this work showed the possibility of representing synchronous systems by interleaved guarded actions and a transformation for the specifications to use existing tools based on a different MoC for verification. This representation was evaluated by implementing a tool that translates AIF files to SAL models. The powerful input language for this tool set allowed the representation of synchronous systems in different ways. These different representations were compared, showing that the representation of synchronous guarded actions by interleaved guarded actions does slow down the verification task in SAL and that the approach used to connect other verification tools by representing the behavior as equations leads to better results. Nevertheless, I think the presented results encourage the modification of the introduced representations such that either new representations are defined or that one or more representations are joint such that e.g. parts of a synchronous system is represented by interleaved guarded actions and the rest is represented by equations.

Representing a synchronous system by other MoC was possible, but the results indicate that this step might not be beneficial for the verification task. Nevertheless, to conclusively determine this, additional work is required that adapt the presented transformation for other tools/languages. These tools should be based on different algorithms and/or differ in the modeling of interleaved guarded actions, because this was not part of this thesis.

References

[AAHM99] R. Alur, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. **Automating modular verification**. In J.C.M. Baeten and S. Mauw, editors, *Concurrency Theory (CONCUR)*, volume 1664 of *LNCS*, pages 82–97, Eindhoven, The Netherlands, 1999. Springer.

[ABKV03] R. Armoni, D. Bustan, O. Kupferman, and M.Y. Vardi. **Resets vs. aborts in linear temporal logic**. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *LNCS*, pages 65–80, Warsaw, Poland, 2003. Springer.

[Abri10] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

[ACHL09] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt. **Control dependence for extended finite state machines**. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 5503 of *LNCS*, pages 216–230, York, England, UK, 2009. Springer.

[Aker78] S.B. Akers. **Binary decision diagrams**. *IEEE Transactions on Computers (T-C)*, C-27(6):509–516, June 1978.

[AlHe99] R. Alur and T.A. Henzinger. **Reactive modules**. *Formal Methods in System Design (FMSD)*, 15(1):7–48, July 1999.

[Andr81a] G.R. Andrews. **Parallel programs: proofs, principles, and practice**. *Communications of the ACM (CACM)*, 24(3):140–145, March 1981.

[ApOl97] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, 2 edition, 1997.

[Apt81] K.R. Apt. **Ten years of Hoare's logic: A survey-part I**. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(4):431–483, 1981.

[Arvi03] Arvind. **Bluespec: A language for hardware design, simulation, synthesis and verification invited talk**. In *Formal Methods and Models for*

Codesign (MEMOCODE), pages 249–254, Mont Saint-Michel, France, 2003. IEEE Computer Society.

[AsMa71] E.A. Ashcroft and Z. Manna. **The translation of 'go to' programs to 'while' programs**. Technical Report CS-TR-71-188, Department of Computer Science, University of California, Stanford, California, USA, January 1971.

[BaBCR10] J. Barnat, L. Brim, M. Ceska, and P. Rockai. **DiVinE: Parallel Distributed Model Checker (Tool paper)**. In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*, pages 4–7. IEEE, 2010.

[BaBS10] D. Baudisch, J. Brandt, and K. Schneider. **Multithreaded code from synchronous programs: Extracting independent threads for OpenMP**. In *Design, Automation and Test in Europe (DATE)*, pages 949–952, Dresden, Germany, 2010. EDA Consortium.

[BaBS11b] D. Baudisch, J. Brandt, and K. Schneider. **Translating synchronous systems to data-flow process networks**. In S.-S. Yeo, B. Vaidya, and G.A. Papadopoulos, editors, *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 354–361, Gwangju, Korea, 2011. IEEE Computer Society.

[BaBS12b] Y. Bai, J. Brandt, and K. Schneider. **Preservation of LTL properties in desynchronized systems**. In S. Shukla, L. Carloni, D. Kroening, and J. Brandt, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 53–63, Arlington, Virginia, USA, 2012. ACM.

[BaLS10] A. Bauer, M. Leucker, and C. Schallhart. **Comparing LTL semantics for runtime verification**. *Journal of Logic and Computation*, 20(3):651–674, February 2010.

[BaPR02] T. Ball, A. Podelski, and S.K. Rajamani. **Relative completeness of abstraction refinement for software model checking**. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2280 of *LNCS*, pages 158–172, Grenoble, France, 2002. Springer.

[Baue12] K. Bauer. *A New Modelling Language for Cyber-physical Systems.* PhD thesis, Department of Computer Science, University of Kaiserslautern, Germany, Kaiserslautern, Germany, January 2012. PhD.

[BBCF00] N.S. Bjørner, A. Browne, M.A. Colón, B. Finkbeiner, Z. Manna, H.B. Sipma, and T.E. Uribe. **Verifying temporal properties of reactive systems: A STeP tutorial**. *Formal Methods in System Design (FMSD)*, 16(3):227–270, June 2000.

[BBGG85] A. Benveniste, P. Bournai, T. Gautier, and P. Le Guernic. **SIGNAL: A data flow oriented language for signal processing**. Research Report 378, Institut

National de Recherche en Informatique et en Automatique (INRIA), Rennes, France, March 1985.

[BCCS03a] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. **Bounded model checking**. In M. Zelkowitz, editor, *Advances in Computers*, volume 58, pages 118–149. Academic Press, 2003.

[BCCZ99] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. **Symbolic model checking without BDDs**. In R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *LNCS*, pages 193–207, Amsterdam, The Netherlands, 1999. Springer.

[BCEH03] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. **The synchronous languages twelve years later**. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[BCMD92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. **Symbolic model checking: 10^{20} states and beyond**. *Information and Computation*, 98(2):142–170, June 1992.

[BCSVZ08] N. Benes, I. Cerna, J. Sochor, P. Varekova, and B. Zimmerova. **A case study in parallel verification of component-based systems**. *Electronic Notes in Theoretical Computer Science*, 220(2):67 – 83, 2008. Proceedings of the 7th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2008).

[BeCC98] S. Berezin, S.V.A. Campos, and E.M. Clarke. **Compositional reasoning in model checking**. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference (COMPOS)*, volume 1536 of *LNCS*, pages 81–102, Bad Malente, Germany, 1998. Springer.

[BeCK10] Nikola Benes, Ivana Cerna, and Milan Krivanek. **Coindivine: Parallel distributed model checker for component-based systems**. In Jiri Barnat and Keijo Heljanko, editors, *PDMC*, volume 72 of *EPTCS*, pages 63–67, 2011.

[BeCo85] G. Berry and L. Cosserat. **The Esterel synchronous programming language and its mathematical semantics**. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency (CONCUR)*, volume 197 of *LNCS*, pages 389–448, Pittsburgh, Pennsylvania, USA, 1985. Springer.

[BeGo92] G. Berry and G. Gonthier. **The Esterel synchronous programming language: Design, semantics, implementation**. *Science of Computer Programming*, 19(2):87–152, 1992.

[BeGu91] A. Benveniste and P. Le Guernic. **Synchronous programming with events and relations: the SIGNAL language and its semantics**. *Science of Computer Programming*, 16(2):103–149, September 1991.

[Berr00] G. Berry. **The Esterel v5 language primer**, July 2000.

[Berr97a] G. Berry. **The Esterel v5 language primer**. http://www.inria.fr/meije/esterel/, April 1997.

[Berr99] G. Berry. **The constructive semantics of pure Esterel**, July 1999.

[BGSS12] J. Brandt, M. Gemünde, K. Schneider, S.K. Shukla, and J.-P. Talpin. **Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions**. *Design Automation for Embedded Systems (DAEM)*, July 2012. DOI 10.1007/s10617-012-9087-9.

[BGSS13] J. Brandt, M. Gemünde, K. Schneider, S.K. Shukla, and J.-P. Talpin. **Embedding polychrony into synchrony**. *IEEE Transactions on Software Engineering (TSE)*, 39(7):917–929, July 2013.

[BiGa96] D.W. Binkley and K.B. Gallagher. **Program slicing**. In M. Zelkowitz, editor, *Advances in Computers*, volume 43, pages 1–50. Academic Press, 1996.

[BoHR97] F.S. de Boer, U. Hannemann, and W.-P. de Roever. **Hoare-style compositional proof systems for reactive shared variable concurrency**. In S. Ramesh and G. Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 1346 of *LNCS*, pages 267–283, Kharagpur, India, 1997. Springer.

[BoJa66] C. Boehm and G. Jacopini. **Flow diagrams, Turing machines, and languages with only two formation rules**. *Communications of the ACM (CACM)*, 9(5):366–371, 1966.

[BoRo98] F.S. de Boer and W.-P. de Roever. **Compositional proof methods for concurrency: A semantic approach**. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference (COMPOS)*, volume 1536 of *LNCS*, pages 632–646, Bad Malente, Germany, 1998. Springer.

[Boul92a] R.J. Boulton. **A HOL semantics for a subset of ELLA**. Technical Report 254, University of Cambridge, Computer Laboratory, April 1992.

[Brad11] A.R. Bradley. **SAT based model checking without unrolling**. In R. Jhala and D.A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 6538 of *LNCS*, pages 70–87, Austin, Texas, USA, 2011. Springer.

[Bran13] J. Brandt. **Synchronous models for embedded software**. Master's thesis, Department of Computer Science, University of Kaiserslautern, July 2013. Habilitation.

[BrGS10] J. Brandt, M. Gemünde, and K. Schneider. **From synchronous guarded actions to SystemC**. In M. Dietrich, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 187–196, Dresden, Germany, 2010. Fraunhofer Verlag.

[Broo85] S.D. Brookes. **On the axiomatic treatment of concurrency**. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*

(CONCUR), volume 197 of *LNCS*, pages 1–34, Pittsburgh, Pennsylvania, USA, 1985. Springer.

[BrSc08a] J. Brandt and K. Schneider. **Formal reasoning about causality analysis**. In O. Ait Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 5170 of *LNCS*, pages 118–133, Montréal, Québec, Canada, 2008. Springer.

[BrSc09] J. Brandt and K. Schneider. **Separate compilation for synchronous programs**. In H. Falk, editor, *Software and Compilers for Embedded Systems (SCOPES)*, volume 320 of *ACM International Conference Proceeding Series*, pages 1–10, Nice, France, 2009. ACM.

[BrSc11] J. Brandt and K. Schneider. **Round trip to asynchrony and synchrony**. In F. Oppenheimer, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 239–248, Oldenburg, Germany, 2011. OFFIS-Institut für Informatik.

[BrSE12] J. Brandt, K. Schneider, and S.A. Edwards. **Translating SHIM to guarded actions**. Technical Report 387/12, University of Kaiserslautern, Kaiserslautern, Germany, February 2012.

[BrSt72] J. Bruno and K. Steiglitz. **The expression of algorithms by charts**. *Journal of the ACM (JACM)*, 19(3):517–525, July 1972.

[Brya86] R.E. Bryant. **Graph-based algorithms for Boolean function manipulation**. *IEEE Transactions on Computers (T-C)*, 35(8):677–691, August 1986.

[CaCo09] J. Casanova and J. Cortadella. **Multi-level clustering for clock skew optimization**. In *International Conference on Computer-Aided Design (ICCAD)*, pages 547–554, San Jose, California, USA, 2009. ACM/IEEE Computer Society.

[CaMS01] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. **Theory of latency-insensitive design**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 20(9):1059–1076, 2001.

[CBKT13] M. Colange, S. Baarir, F. Kordon, and Y. Thierry-Mieg. **Towards distributed software model-checking using decision diagrams**. In N. Sharygina and H. Veith, editors, *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 830–845, Saint Petersburg, Russia, 2013. Springer.

[CCGO04] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. **Efficient verification of sequential and concurrent C programs**. *Formal Methods in System Design (FMSD)*, 25(2-3):129–166, September 2004.

[CEFJ96] E.M. Clarke, R. Enders, T. Filkorn, and S. Jha. **Exploiting symmetry in temporal logic model checking**. *Formal Methods in System Design (FMSD)*, 9(1-2):77–104, 1996.

[CFRR99] E.M. Clarke, M. Fujita, S.P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. **Program slicing of hardware description languages**. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME)*, volume 1703 of *LNCS*, pages 298–313, Bad Herrenalb, Germany, 1999. Springer.

[CGJL03] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. **Counterexample-guided abstraction refinement for symbolic model checking**. *Journal of the ACM (JACM)*, 50(5):752–794, September 2003.

[ChJa12] D.-H. Chu and J. Jaffar. **A complete method for symmetry reduction in safety verification**. In P. Madhusudan and S.A. Seshia, editors, *Computer Aided Verification (CAV)*, volume 7358 of *LNCS*, pages 616–633, Berkeley, California, USA, 2012. Springer.

[ChMi89] K.M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, Austin, Texas, USA, May 1989.

[ChMP92] E.Y. Chang, Z. Manna, and A. Pnueli. **Characterization of temporal property classes**. In W. Kuich, editor, *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 623 of *LNCS*, pages 474–486, Vienna, Austria, 1992. Springer.

[CiGr12] A. Cimatti and A. Griggio. **Software model checking via IC3**. In P. Madhusudan and S.A. Seshia, editors, *Computer Aided Verification (CAV)*, volume 7358 of *LNCS*, pages 277–293, Berkeley, California, USA, 2012. Springer.

[ClES83] E.M. Clarke, E.A. Emerson, and A.P. Sistla. **Automatic verification of finite-state concurrent systems using temporal logic**. In *Principles of Programming Languages (POPL)*, pages 117–126, 1983.

[ClGL92] E.M. Clarke, O. Grumberg, and D.E. Long. **Model checking and abstraction**. In *Principles of Programming Languages (POPL)*, pages 342–354, Albuquerque, New Mexico, USA, 1992. ACM.

[ClGL94a] E.M. Clarke, O. Grumberg, and D.E. Long. **Model checking and abstraction**. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, September 1994.

[ClGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 1999.

[ClLM89a] E.M. Clarke, D.E. Long, and K.L. McMillan. **Compositional model checking**. In *Logic in Computer Science (LICS)*, pages 353–361, Washington, District of Columbia, USA, 1989. IEEE Computer Society.

[Cous90] P. Cousot. **Methods and logics for proving programs**. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 15, pages 841–993. Elsevier, 1990.

[Darr79] J.A. Darringer. **The application of program verification techniques to hardware verification**. In D.W. Hightower, editor, *Design Automation Conference (DAC)*, pages 375–381, San Diego, California, USA, 1979. ACM.

[Dijk68] E.W. Dijkstra. **Go to statement considered harmful**. *Communications of the ACM (CACM)*, 11(3):147–148, 1968.

[Dijk72] E.W. Dijkstra. **Notes on structured programming**. In O.-J. Dahl, E.W. Disjkstra, and C.A.R. Hoare, editors, *Structured Programming.* Academic Press, 1972.

[Dijk75] E.W. Dijkstra. **Guarded commands, nondeterminacy and formal derivation of programs**. *Communications of the ACM (CACM)*, 18(8):453–457, 1975.

[Dill96] D.L. Dill. **The Murphi verification system**. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 390–393, New Brunswick, New Jersey, USA, 1996. Springer.

[Ding00] J. Dingel. **Towards a unified development methodology for shared-variable parallel and distributed programs**. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Integrated Formal Methods (IFM)*, volume 1945 of *LNCS*, pages 214–234, Dagstuhl, Germany, 2000. Springer.

[Elgo76] C.C. Elgot. **Structured programming with and without 'go to' statements**. *IEEE Transactions on Software Engineering (T-SE)*, SE-2(1):41–54, March 1976.

[EmCl80] E.A. Emerson and E.M. Clarke. **Characterizing correctness properties of parallel programs using fixpoints**. In J.W. de Bakker and J. van Leeuwen, editors, *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 85 of *LNCS*, pages 169–181, Noordweijkerhout, The Netherlands, 1980. Springer.

[Emer90] E.A. Emerson. **Temporal and modal logic**. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 16, pages 995–1072. Elsevier, 1990.

[EmSi96] E.A. Emerson and A.P. Sistla. **Symmetry and model checking**. *Formal Methods in System Design (FMSD)*, 9:105–131, 1996.

[EMSS91] E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinivasan. **Quantitative temporal reasoning**. In E.M. Clarke and R.P. Kurshan, editors, *Computer Aided Verification (CAV)*, volume 531 of *LNCS*, pages 136–145, New Brunswick, New Jersey, USA, 1991. Springer.

[Este04] **Esterel technologies: Scade language reference manual. (2004)**. `http://www.esterel-technologies.com/products/scade-system/`.

[FaFM10] Y. Falcone, J.-C. Fernandez, and L. Mounier. **What can you verify and enforce at runtime?**. Research Report TR-2010-5, Verimag, January 2010.

[FGHH05] L. Fix, O. Grumberg, A. Heyman, T. Heyman, and A. Schuster. **Verifying very large industrial circuits using 100 processes and beyond**. In D.A. Peled and Y.-K. Tsay, editors, *Automated Technology for Verification and Analysis (ATVA)*, volume 3707 of *LNCS*, pages 11–25, Taipei, Taiwan, 2005. Springer.

[Floy67] R.W. Floyd. **Assigning meanings to programs**. In J.T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, USA, 1967. American Mathematical Society.

[GaGu89] S.J. Garland and J.V. Guttag. **An overview of LP, the larch power**. In N. Dershowitz, editor, *Rewriting Techniques and Applications (RTA)*, volume 355 of *LNCS*, pages 137–151, Chapel Hill, North Carolina, USA, 1989. Springer.

[GeBS14] M. Gesell, F. Bichued, and K. Schneider. **Using different representations of synchronous systems in SAL**. In J. Ruf, D. Allmendinger, and M. Michel, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 13–24, IBM Deutschland, Böblingen, Germany, 2014. Cuvillier.

[GeMS13] M. Gesell, A. Morgenstern, and K. Schneider. **Lifting verification results for preemption statements**. In R.M. Hierons, M.G. Merayo, and M. Bravetti, editors, *Software Engineering and Formal Methods (SEFM)*, volume 8137 of *LNCS*, pages 91–105, Madrid, Spain, 2013. Springer.

[Gemu13a] M. Gemünde. *Clock Refinement in Imperative Synchronous Programs*. PhD thesis, Department of Computer Science, University of Kaiserslautern, Germany, Kaiserslautern, Germany, October 2013. PhD.

[GeSc12] M. Gesell and K. Schneider. **A Hoare calculus for the verification of synchronous languages**. In K. Claessen and N. Swamy, editors, *Programming Languages meets Program Verification (PLPV)*, pages 37–48, Philadelphia, Pennsylvania, USA, 2012. ACM.

[GeSc12a] M. Gesell and K. Schneider. **Interactive verification of synchronous systems**. In S. Shukla, L. Carloni, D. Kroening, and J. Brandt, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 75–84, Arlington, Virginia, USA, 2012. ACM.

[GeSc13a] M. Gesell and K. Schneider. **Modular verification of synchronous programs**. In J. Carmona Mihai, T. Lazarescu, and M. Pietkiewicz-Koutny, editors, *Application of Concurrency to System Design (ACSD)*, pages 70–79, Barcelona, Spain, 2013. IEEE Computer Society.

[GeSc13c] M. Gesell and K. Schneider. **Translating synchronous guarded actions to interleaved guarded actions**. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 167–176, Portland, OR, USA, 2013. IEEE Computer Society.

[Gira05] A. Girard. **Reachability of uncertain linear systems using zonotopes**. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume 3414 of *LNCS*, pages 291–305, Zurich, Switzerland, 2005. Springer.

[Gode03] P. Godefroid. **Reasoning about abstract open systems with generalized module checking**. In R. Alur and I. Lee, editors, *Embedded Software (EMSOFT)*, volume 2855 of *LNCS*, pages 223–240, Philadelphia, Pennsylvania, USA, 2003. Springer.

[Gode05] P. Godefroid. **Software model checking: The VeriSoft approach**. *Formal Methods in System Design (FMSD)*, 26(2):77–101, March 2005.

[Gode95] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State Explosion Problem.* PhD thesis, Université de Liege, Institut Montefiore, 1995. PhD.

[GoMW79] M.J.C. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF: A mechanized logic of computation*, volume 78 of *LNCS*. Springer, 1979.

[Gord00] M.J.C. Gordon. **Reachability programming in HOL98 using BDDs**. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 1869 of *LNCS*, pages 179–196, Portland, Oregon, USA, 2000. Springer.

[Gord86] M.J.C. Gordon. **Why higher-order logic is a good formalism for specifying and verifying hardware**. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. North-Holland, Cambridge, England, UK, 1986.

[Gord95] M.J.C. Gordon. **The semantic challenge of Verilog HDL**. In *Logic in Computer Science (LICS)*, pages 136–145, San Diego, California, USA, 1995. IEEE Computer Society.

[Gord98] M.J.C. Gordon. **Event and cycle semantics of hardware description languages**. http://www.cl.cam.ac.uk/users/mjcg/Verilog, January 1998.

[Grie81] D. Gries. *The Science of Programming.* Springer, 1981.

[GrLo91] O. Grumberg and D.E. Long. **Model checking and modular verification**. In J.C.M. Baeten and J.F. Groote, editors, *Concurrency Theory (CONCUR)*, volume 527 of *LNCS*, pages 250–265, Amsterdam, The Netherlands, 1991. Springer.

[GrVe08] O. Grumberg and H. Veith, editors. *25 Years of Model Checking – History, Achievements, Perspectives*, volume 5000 of *LNCS*. Springer, 2008.

[GrZa03] J.F. Groote and H. Zantema. **Resolution and binary decision diagrams cannot simulate each other polynomially**. *Discrete Applied Mathematics*, 130(2):157–171, August 2003.

[Gupt92] A. Gupta. **Formal hardware verification methods: A survey**. *Formal Methods in System Design (FMSD)*, 1(2-3):151–238, 1992.

[HaBS12] Z. Hassan, A.R. Bradley, and F. Somenzi. **Incremental, inductive CTL model checking**. In P. Madhusudan and S.A. Seshia, editors, *Computer Aided*

Verification (CAV), volume 7358 of *LNCS*, pages 532–547, Berkeley, California, USA, 2012. Springer.

[Halb05] N. Halbwachs. **A synchronous language at work: the story of Lustre**. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 3–11, Verona, Italy, 2005. IEEE Computer Society.

[Halb93] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer, 1993.

[HaLR93] N. Halbwachs, F. Lagnier, and P. Raymond. **Synchronous observers and the verification of reactive systems**. In *Algebraic Methodology and Software Technology (AMAST)*, Workshops in Computing, pages 83–96, Enschede, The Netherlands, 1993. Springer.

[HaPn85] D. Harel and A. Pnueli. **On the development of reactive systems**. In K.R. Apt, editor, *Logic and Models of Concurrent Systems*, pages 477–498. Springer, 1985.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. **The synchronous dataflow programming language LUSTRE**. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HeHK01] T.A. Henzinger, B. Horowitz, and C. Meyer Kirsch. **Giotto: A time-triggered language for embedded programming**. In T.A. Henzinger and C. Meyer Kirsch, editors, *Embedded Software (EMSOFT)*, volume 2211 of *LNCS*, pages 166–184, Tahoe City, California, USA, 2001. Springer.

[HeQR00] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. **Decomposing refinement proofs using Assume-Guarantee reasoning**. In E. Sentovich, editor, *International Conference on Computer-Aided Design (ICCAD)*, pages 245–252, San Jose, California, USA, 2000. ACM/IEEE Computer Society.

[HJMM04] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. **Abstractions from proofs**. In N.D. Jones and X. Leroy, editors, *Principles of Programming Languages (POPL)*, pages 232–244, Venice, Italy, 2004. ACM.

[HoAr04] J.C. Hoe and Arvind. **Operation-centric hardware description and synthesis**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 23(9):1277–1288, September 2004.

[Hoar69] C.A.R. Hoare. **An axiomatic basis for computer programming**. *Communications of the ACM (CACM)*, 12(10):576–580, 1969.

[Hoar78] C.A.R. Hoare. **Communicating sequential processes**. *Communications of the ACM (CACM)*, 21(8):666–677, 1978.

[Jant04] A. Jantsch. *Modeling Embedded Systems and SoCs.* Morgan Kaufmann, 2004.

[JoSe94] J.J. Joyce and C.-J.H. Seger. **The HOL-Voss system: Model-checking inside a general-purpose theorem-prover**. In J.J. Joyce and C.-J.H. Seger, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 780 of *LNCS*, pages 185–198, Vancouver, British Columbia, Canada, 1994. Springer.

[KaRa88] R.M. Karp and V. Ramachandran. **A survey of parallel algorithms for shared-memory machines**. Technical Report UCB/CSD-88-408, EECS Department, University of California, Berkeley, http://www.eecs.berkeley.edu/Pubs/TechRpts/1988/CSD-88-408.pdf, March 1988.

[KeGr99] C. Kern and M.R. Greenstreet. **Formal verification in hardware design: A survey**. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999.

[KoFu06] H. Kondoha and K. Futatsugi. **To use or not to use the goto statement: Programming styles viewed from Hoare logic**. *Science of Computer Programming*, 60:82–116, 2006.

[Kosa73] S.R. Kosaraju. **Analysis of structured programs**. In *Symposium on Theory of Computing (STOC)*, pages 240–252, Austin, Texas, USA, 1973. ACM.

[Kosa76] S.R. Kosaraju. **On structuring flowcharts (preliminary version)**. In *Symposium on Theory of Computing (STOC)*, pages 101–111, Hershey, Pennsylvania, USA, 1976. ACM.

[KoVo13] L. Kovács and A. Voronkov. **First-order theorem proving and vampire**. In N. Sharygina and H. Veith, editors, *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 1–35, Saint Petersburg, Russia, 2013. Springer.

[KuLa93] R.P. Kurshan and L. Lamport. **Verification of a multiplier: 64 bits and beyond**. In C. Courcoubetis, editor, *Computer Aided Verification (CAV)*, volume 697 of *LNCS*, pages 166–179, Elounda, Crete, Greece, 1993. Springer.

[KuSK93a] R. Kumar, K. Schneider, and T. Kropf. **Structuring and automating hardware proofs in a higher-order theorem-proving environment**. *Formal Methods in System Design (FMSD)*, 2(2):165–230, 1993.

[KuVa95] O. Kupferman and M.Y. Vardi. **On the complexity of branching modular model checking (extended abstract)**. In I. Lee and S.A. Smolka, editors, *Concurrency Theory (CONCUR)*, volume 962 of *LNCS*, pages 408–422, Philadelphia, Pennsylvania, USA, 1995. Springer.

[KuVa96] O. Kupferman and M.Y. Vardi. **Module checking**. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 75–86, New Brunswick, New Jersey, USA, 1996. Springer.

[KuVW01] O. Kupferman, M.Y. Vardi, and P. Wolper. **Module checking**. *Information and Computation*, 164:322–344, 2001.

[LaGr98] K. Laster and O. Grumberg. **Modular model checking of software**. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of*

Systems (TACAS), volume 1384 of *LNCS*, pages 20–35, Lisbon, Portugal, 1998. Springer.

[Lamp80a] L. Lamport. **The ‘ Hoare logic’ of concurrent programs**. *Acta Informatica*, 14:21–37, 1980.

[LaSc84] L. Lamport and F.B. Schneider. **The ‘ Hoare’ logic CSP, and all that**. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):281–296, April 1984.

[LBHJ04] T. Latvala, A. Biere, K. Heljanko, and T. Junttila. **Simple bounded LTL model checking**. In A.J. Hu and A.K. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, volume 3312 of *LNCS*, pages 186–200, Austin, Texas, USA, 2004. Springer.

[LeSa98] E.A. Lee and A. Sangiovanni-Vincentelli. **A framework for comparing models of computation**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 17(12):1217–1229, December 1998.

[LGSB95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. **Property preserving abstractions for the verification of concurrent systems**. *Formal Methods in System Design (FMSD)*, 6:11–44, February 1995.

[Long93] D.E. Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, Carnegie Mellon University, 1993. PhD.

[Mali94] S. Malik. **Analysis of cycle combinational circuits**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 13(7):950–956, July 1994.

[MaPn92] Z. Manna and A. Pnueli. *The temporal Logic of Reactive and Concurrent Systems.* Springer, 1992.

[MaPn95b] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems – Safety.* Springer, 1995.

[MaTu89] A.R. Martin and J.V. Tucker. **The concurrent assignment representation of synchronous systems**. *Parallel Computing*, 9(2):227–256, 1989.

[McMi03] K.L. McMillan. **Interpolation and SAT-based model checking**. In W.A. Hunt and F. Somenzi, editors, *Computer Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 1–13, Boulder, Colorado, USA, 2003. Springer.

[McMi03a] K.L. McMillan. **Methods for exploiting SAT solvers in unbounded model checking**. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 135–144, Mont Saint-Michel, France, 2003. IEEE Computer Society.

[McMi92a] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, May 1992. PhD.

[McMi93a] K.L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.

[McMi98] K.L. McMillan. **Verification of an implementation of Tomasulo's algorithm by compositional model checking**. In A.J. Hu and M.Y. Vardi, editors, *Computer Aided Verification (CAV)*, volume 1427 of *LNCS*, pages 110–121, Vancouver, British Columbia, Canada, 1998. Springer.

[McMi99a] K.L. McMillan. **Verification of infinite state systems by compositional model checking**. Technical report, Cadence Research Labs, Berkeley, California, USA, 1999.

[McMi99b] K.L. McMillan. **A methodology for hardware verification using compositional model checking**. Technical report, Cadence Research Labs, Berkeley, California, USA, 1999.

[McQS00] K.L. McMillan, S. Qadeer, and J.B. Saxe. **Induction in compositional model checking**. In E.A. Emerson and A.P. Sistla, editors, *Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 312–327, Chicago, Illinois, USA, 2000. Springer.

[Melh89a] T. Melham. **Automating recursive type definitions in higher order logic**. In G. Birtwistle and P.A. Subramanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer, 1989.

[Merc96] A. Merceron. **Checking synchronous programs using automatic abstraction, modular verification, and assumption discharge**. GMD, Schloß Birlinghoven, Sankt Augustin, Germany, 1996.

[MeSt95] N. Mellergaard and J. Staunstrup. **Tutorial on design verification with synchronized transitions**. In R. Kumar and T. Kropf, editors, *Theorem Provers in Circuit Design (TPCD)*, volume 901 of *LNCS*, pages 239–257, Bad Herrenalb, Germany, 1995. Springer.

[Meta97] P.T. Metaxas. **Thinking and programming in parallel**, 1997. Lecture Notes.

[Mill75] H.D. Mills. **The new math of computer programming**. *Communications of the ACM (CACM)*, 18(1):43–48, January 1975.

[MoGS12] A. Morgenstern, M. Gesell, and K. Schneider. **An asymptotically correct finite path semantics for LTL**. In N. Bjørner and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 7180 of *LNCS*, pages 304–319, Mérida, Venezuela, 2012. Springer.

[MoPS11] G. Morbé, F. Pigorsch, and C. Scholl. **Fully symbolic model checking for timed automata**. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification (CAV)*, volume 6806 of *LNCS*, pages 616–632, Snowbird, Utah, USA, 2011. Springer.

[MORR04] L.M. de Moura, S. Owre, H. Rueß, J.M. Rushby, N. Shankar, M. Sorea, and A. Tiwari. **SAL 2**. In R. Alur and D.A. Peled, editors, *Computer Aided*

Verification (CAV), volume 3114 of *LNCS*, pages 496–500, Boston, Massachusetts, USA, 2004. Springer.

[MoRS03] L. de Moura, H. Rueß, and M. Sorea. **Bounded model checking and induction: From refutation to verification**. In W.A. Hunt and F. Somenzi, editors, *Computer Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 14–26, Boulder, Colorado, USA, 2003. Springer.

[MoSL08] A. Morgenstern, K. Schneider, and S. Lamberti. **Generating deterministic ω-automata for most LTL formulas by the breakpoint construction**. In C. Scholl and S. Disch, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 119–128, Freiburg, Germany, 2008. Shaker.

[Moss06] P.D. Mosses. **Formal semantics of programming languages**. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 148:41–73, 2006.

[OwGr76a] S.S. Owicki and D. Gries. **An axiomatic proof technique for parallel programs I**. *Acta Informatica*, 6(4):319–340, 1976.

[OwGr76b] S.S. Owicki and D. Gries. **Verifying properties of parallel programs: An axiomatic approach**. *Communications of the ACM (CACM)*, 19(5):279–284, 1976.

[OwRS92] S. Owre, J.M. Rushby, and N. Shankar. **PVS: A prototype verification system**. In D. Kapur, editor, *Conference on Automated Deduction (CADE)*, volume 607 of *LNCS*, pages 748–752, Saratoga Springs, New York, USA, 1992. Springer.

[Paul94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[Plot81] G.D. Plotkin. **A structural approach to operational semantics**. Technical Report FN-19, DAIMI, Århus, Denmark, 1981.

[Pnue77a] A. Pnueli. **The temporal logic of programs**. In *Foundations of Computer Science (FOCS)*, pages 46–57, Providence, Rhode Island, USA, 1977. IEEE Computer Society.

[PoCB04] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. **Concurrency in synchronous systems**. In *Application of Concurrency to System Design (ACSD)*, pages 67–76, Hamilton, Ontario, Canada, 2004. IEEE Computer Society.

[PoEB07] D. Potop-Butucaru, S.A. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.

[RaSS95] S. Rajan, N. Shankar, and M.K. Srivas. **An integration of model checking with automated proof checking**. In P.L. Wolper, editor, *Computer Aided Verification (CAV)*, volume 939 of *LNCS*, pages 84–97, Liège, Belgium, 1995. Springer.

[RBHH01] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[Reet95] R. Reetz. **Deep embedding VHDL**. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 971 of *LNCS*, pages 277–292, Aspen Grove, Utah, USA, 1995. Springer.

[Roev98] W.-P. de Roever. **The need for compositional proof systems: A survey**. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference (COMPOS)*, volume 1536 of *LNCS*, pages 1–22, Bad Malente, Germany, 1998. Springer.

[SBST05a] K. Schneider, J. Brandt, T. Schüle, and T. Türk. **Improving constructiveness in code generators**. In *Synchronous Languages, Applications, and Programming (SLAP)*, pages 1–19, Edinburgh, Scotland, UK, 2005.

[ScBr08] K. Schneider and J. Brandt. **Performing causality analysis by bounded model checking**. In *Application of Concurrency to System Design (ACSD)*, pages 78–87, Xi'an, China, 2008. IEEE Computer Society.

[ScBS04b] K. Schneider, J. Brandt, and T. Schüle. **Causality analysis of synchronous programs with delayed actions**. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 179–189, Washington, District of Columbia, USA, 2004. ACM.

[ScBS06] K. Schneider, J. Brandt, and T. Schüle. **A verified compiler for synchronous programs with local declarations**. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 153(4):71–97, 2006.

[Schn01a] K. Schneider. **Embedding imperative synchronous languages in interactive theorem provers**. In *Application of Concurrency to System Design (ACSD)*, pages 143–154, Newcastle Upon Tyne, England, UK, 2001. IEEE Computer Society.

[Schn01b] K. Schneider. **Improving automata generation for linear temporal logic by considering the automata hierarchy**. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2250 of *LNCS*, pages 39–54, Havana, Cuba, 2001. Springer.

[Schn02] K. Schneider. **Proving the equivalence of microstep and macrostep semantics**. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 2410 of *LNCS*, pages 314–331, Hampton, Virginia, USA, 2002. Springer.

[Schn03] K. Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms.* Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[Schn09] K. Schneider. **The synchronous programming language Quartz**. Internal Report 375, Department of Computer Science, University of Kaiserslautern,

Kaiserslautern, Germany, December 2009.

[ScHo99] K. Schneider and D.W. Hoffmann. **A HOL conversion for translating linear time temporal logic to omega-automata**. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 1690 of *LNCS*, pages 255–272, Nice, France, 1999. Springer.

[ScKr97a] K. Schneider and T. Kropf. **The C@S system: Combining proof strategies for system verification**. In T. Kropf, editor, *Formal Hardware Verification – Methods and Systems in Comparison*, volume 1287 of *LNCS*, pages 248–329. Springer, August 1997.

[ShSS00] M. Sheeran, S. Singh, and G. Stålmarck. **Checking safety properties using induction and a SAT-solver**. In W.A. Hunt and S.D. Johnson, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *LNCS*, pages 108–125, Austin, Texas, USA, 2000. Springer.

[SiSh07] G. Singh and S.K. Shukla. **Algorithms for low power hardware synthesis from concurrent action oriented specifications CAOS**. *International Journal of Embedded Systems (IJES)*, 3(1/2):83–92, 2007.

[Stau97] J. Staunstrup. **Design verification using Synchronized Transitions**. In T. Kropf, editor, *Formal Hardware Verification – Methods and Systems in Comparison*, volume 1287 of *LNCS*, pages 114–155. Springer, August 1997.

[StCO06] M. Stokelya, S. Chakia, and J. Ouaknine. **Parallel assignments in software model checking**. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 157:77–94, 2006.

[StGG90] J. Staunstrup, S.J. Garland, and J.V. Guttag. **Localized verification of circuit descriptions**. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 349–364, Grenoble, France, 1990. Springer.

[StGG92] J. Staunstrup, S.J. Garland, and J.V. Guttag. **Mechanized verification of circuit descriptions using the Larch Prover**. In V. Stavridou and T.F. Melham, editors, *Theorem Provers in Circuit Design (TPCD)*, pages 277–300. Elsevier, 1992.

[StGr88] J. Staunstrup and M.R. Greenstreet. **From high-level descriptions to VLSI circuits**. *BIT Numerical Mathematics*, 28(3):620–638, 1988.

[StMe95] J. Staunstrup and N. Mellergaard. **Localized verification of modular designs**. *Formal Methods in System Design (FMSD)*, 3(6):295–320, 1995.

[TBGS14] J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, and S. Shukla. **Constructive polychronous systems**. *Science of Computer Programming (SCICO)*, 2014. http://dx.doi.org/10.1016/j.scico.2014.04.009.

[Tip95] F. Tip. **A survey of program slicing techniques**. *Journal of Programming Languages*, 3(3):121–189, 1995.

[Weis79] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, Ann Arbor, MI, USA, 1979. AAI8007856.

[XQZW05] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. **A brief survey of program slicing**. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, March 2005.

[ZhHo92] C. Zhou and C.A.R. Hoare. **A model for synchronous switching circuits and its theory of correctness**. *Formal Methods in System Design (FMSD)*, 1(1):7–28, 1992.

CURRICULUM VITAE

Persönliche Daten

Name	Manuel Gesell
Staatsangehörigkeit	deutsch

Berufserfahrung

seit 07/2014	**Software-Developer TCMS Simulation and Tools**, Bombardier Transportation GmbH in Mannheim
07/2008 – 06/2014	**Wissenschaftlicher Mitarbeiter**, Technische Universität Kaiserslautern Fachbereich Informatik, Arbeitsgruppe Eingebettete Systeme

Akademische Ausbildung

10/2003 – 06/2008	**Studium der Informatik**, Technische Universität Kaiserslautern Anwendungsfach Eingebettete Systeme, Nebenfach Elektrotechnik **Diplomarbeit** mit dem Thema *„Probabilistic Model Checking of Synchronous Programs“* Abschluss: **Diplom-Informatiker (Dipl.-Inf.)**

Schulausbildung

09/1999 – 06/2002	**Allgemeine Hochschulreife** (Note 1,3) Berufliches Gymnasium in Hofheim, Fachrichtung Technikwissenschaft