

Verifying Concurrent Systems with Symbolic Execution

Temporal Reasoning is Symbolic Execution with a Little Induction

Dissertation
zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Angewandte Informatik
der Universität Augsburg
im Jahr 2005 von

Michael Balser



Amtierender Dekan: Prof. Dr. Wolfgang Reif
Gutachter: Prof. Dr. Wolfgang Reif
Prof. Dr. Walter Vogler
Tag der Prüfung: 12. Juli 2005
Prüfer: Prof. Dr. Wolfgang Reif
Prof. Dr. Bernhard Bauer
Prof. Dr. Bernhard Möller

Abstract

Symbolic execution is an intuitive strategy to verify sequential programs, which can be automated to a large extent. We have successfully carried over this method of proof to the interactive verification of concurrent systems. The resulting strategy can be applied to the verification of complex parallel programs and arbitrary (linear) temporal formulas. Our underlying logic is defined such that operators for parallel programs and temporal logic can be arbitrarily nested. We support interleaving with explicit blocking, non-deterministic choice, and others. Most important, the semantics of all of the operators are compositional. Thus, systems can be abstracted and proofs can be decomposed. This ensures that our strategy of proof can be applied to the verification of large, concurrent systems.

Preface

The idea of concurrency is becoming more and more important in computer science, as computers are widely connected over the internet and even the smallest device interacts with other devices to implement complex functionality. For example, in automotive engineering, an increasing number of embedded computers form a concurrent system to control even safety critical parts like airbags, breaks, steering, and others. It is, however, much more difficult to design a concurrent system compared to a sequential program. Instead of a single flow of execution, the control flow is more complex. Particularly, for reactive systems, not only the final result of execution but the sequence of output over time is relevant to system behaviour; additional sources of errors are insufficient synchronisation, deadlocks, livelocks, race conditions, priority issues, and others.

The possibility of finding errors by means of testing strategies is limited, because, especially for interleaved systems, an exponential amount of possible executions must be considered. The execution is nondeterministic, making it difficult to reproduce errors. An alternative to testing is the use of formal methods to specify and verify concurrent systems with mathematical rigour. In practice, automatic methods – especially model checking – are successfully applied to discover flaws in the design and implementation of systems. Part of the success of model checking is due to the fact that it can be applied by software engineers without being experts in formal methods. However, automatic strategies are limited to small and medium sized state finite applications. Large and state infinite systems must be manually abstracted to ensure that formal analysis terminates. Manual abstraction is difficult and calls for an expert with formal background; the soundness of the abstraction must be ensured.

Instead of automatic algorithms, large and state infinite systems can be analysed with interactive verification. Existing calculi to reason about concurrent systems are generally difficult to apply. The strategy of symbolic execution, on the other hand, has been successfully applied to the interactive verification of sequential programs. Symbolic execution gives intuitive proofs with a high degree of automation, and can therefore be applied, to some extent, by non-experts. Our idea is to carry over symbolic execution to the interactive verification of temporal properties of concurrent systems.

Related work This work is inspired by the following existing approaches.

Using *Temporal Logic of Actions* (TLA) [18], concurrent systems can be combined with simple logical conjunction [1]. Conjunction of systems is compositional and proofs can be decomposed! However, system models in TLA are restricted to simple state transition systems; complex programming constructs like loops, conditionals, interleaving, synchronous execution, etc. must be translated into basic transitions with additional program counters. The interactive calculus of [18] to verify temporal properties of systems in TLA is very basic, and it is difficult to automate part of the deduction.

The *Stanford Temporal Prover* (STeP) [6] offers a rich input language supporting interleaved parallel processes, message passing, and other features. The systems are automatically translated into fair transition systems to which model checking or deductive reasoning can be applied. Especially the use of verification diagrams as a high level abstraction of proofs in temporal logic is of interest. However, verification diagrams in practice are restricted to the verification of a certain class of temporal properties. We hope that symbolic execution is more automatic than verification in STeP and that the strategy can be efficiently applied to any type of temporal property. Furthermore, we try to directly support the original system model within the interactive calculus instead of translating the model to a (fair) transition system.

Our temporal logic is based on *Interval Temporal Logic* (ITL) [22] which is a linear time logic and combines temporal formulas and program operators within the same formalism. We have enriched the set of program operators by interleaving, nondeterministic choice, and others. To some extent, we also consider branching time logics, e.g., *Computation Tree Logic* (CTL) [8].

Early parts of this work have been integrated into the Verification Support Environment (VSE) [16]. A more advanced implementation of the proof method with direct support for parallel programs with interleaving and advanced heuristics for automation is available in the KIV system [27] [4].

Overview Chapter 1 gives an extended overview of our approach. Syntax and semantics of our logic is formally defined in Chapter 2. The recipe for interactive verification of concurrent systems with symbolic execution is outlined in Chapter 3. It consists of four ingredients: (i) symbolic execution which is described in Chapter 4, (ii) sequencing to reduce the number of execution paths to be considered in a proof (see Chapter 5), (iii) induction in Chapter 6, and (iv) abstraction to structure proofs for large systems (Chapter 7). Chapter 8 contains a comparison, summarises open issues and concludes. A list of calculus rules are part of the Appendices A and B. Proofs have been put into Appendix C unless they directly contribute to the understanding of the material. Finally, Appendix D contains a summary of mathematical symbols.

Acknowledgements I would like to thank Prof. Wolfgang Reif for his considerable support and motivation, Dr. Gerhard Schellhorn for fruitful discussions and stimulations, Christoph Duelli, Jonathan Schmitt, Simon Bäumler, Frank Ortmeier, and Andreas

Thums, members of our working group who were directly involved in the interactive verification of concurrent systems and who provided valuable feedback and ideas, Thomas Sorg who defined first steps for this approach in his master thesis, Florian Nafz, Nadhem Kachroudi, and Alwin Hoffmann, student workers who implemented and evaluated parts of the strategy, all partners of the Procure project where the proof method has been applied to the verification of medical guidelines, Werner Balser for proof reading, and Karin Balser for her loving support and understanding of a crazy logician.

Contents

Preface	v
1 Motivation	1
1.1 Example	1
1.2 Syntax	3
1.3 Semantics	5
1.4 Calculus	5
1.5 Ingredient 1 – Symbolic Execution	6
1.6 Ingredient 2 – Sequencing	12
1.7 Ingredient 3 – Induction	13
1.8 Ingredient 4 – Abstraction	15
1.9 Outlook	19
2 Syntax and Semantics	21
2.1 Syntax	21
2.2 Semantics	27
2.3 Synchronisation	37
2.4 Interleaving	38
2.5 SOS notation	42
2.6 Interleaving intervals	44
2.7 Dijkstra’s choice operator	47
2.8 Atomic steps	48
2.9 Labels	49

2.10	Restricted expressions	49
2.11	Substitution	50
2.12	Conclusion	57
3	Calculus	59
3.1	Existing approaches	60
3.2	Rewriting with context	63
3.3	Example: Propositional Logic	76
3.4	Conclusion	83
4	Ingredient 1 – Symbolic Execution	85
4.1	Idea	85
4.2	Normal form	86
4.3	Step	90
4.4	Symbolic Execution	91
4.5	Sequential programs	92
4.6	Excursion: Classification of operators	97
4.7	Parallel programs	98
4.8	Local variables and quantifiers	101
4.9	Procedure calls	102
4.10	Example: Executing parallel programs	104
4.11	Temporal logic operators	105
4.12	Atomic formulas	107
4.13	Propositional operators	107
4.14	Example: Executing temporal formulas	109
4.15	Example: Semaphore (part 1)	111
4.16	Conclusion	116

5 Ingredient 2 – Sequencing	119
5.1 Proof graph	119
5.2 Sequencing	120
5.3 Example: Semaphore (part 2)	123
5.4 Conclusion	126
6 Ingredient 3 – Induction	127
6.1 Basic induction	127
6.2 Induction with liveness	129
6.3 Extracting liveness	130
6.4 Example: Semaphore (part 3)	136
6.5 Automation	143
6.6 Completeness	143
6.7 Summary	143
7 Ingredient 4 – Abstraction	145
7.1 Congruence rules	145
7.2 Compositional operational semantics	146
7.3 Example: Handshaking	147
7.4 Verification diagrams	154
7.5 Conclusion	156
8 Concluding Remarks	157
8.1 Comparison	159
8.2 Outlook	162
A Rewriting	165
A.1 Basic rules	165
A.2 Rewrite rules	166
A.3 Congruence rules	167
A.4 Congruence rules with additional context	168
A.5 Congruence rules with restricted context	170

B Operators	173
B.1 Propositional Logic	174
B.2 First Order Logic	177
B.3 Transitions	180
B.4 System Operators	185
B.5 ITL Operators	187
B.6 LTL Operators	188
B.7 Program Quantifiers	190
B.8 Sequential Programs	191
B.9 Synchronisation	195
B.10 Interleaving	195
B.11 Nondeterministic choice	199
B.12 Atomic steps	200
B.13 Labels	201
B.14 Procedures	201
C Proofs	203
C.1 Semantics	203
C.2 Execution	212
C.3 Induction	214
D Mathematical Symbols	223
Bibliography	225

Chapter 1

Motivation

This chapter is designed to give an overview of our approach. It motivates the necessity of an improved calculus for verifying concurrent systems and explains the different ideas of the method of proof. The chapter is centered around an example. Formal notations are only introduced where necessary and details are abstracted. Calculus rules may be restricted to special cases and may even be wrong in general (but sound for the special case). Details are left to following chapters.

1.1 Example

Example Figure 1.1 gives a concurrent system from [28]. The system consists of three processes running in parallel, the producer, the channel and the consumer. The producer produces data i and sends the data to the channel. Both communicate using a shared variable c_1 . The channel buffers the data in b and asynchronously forwards it to the consumer (variable c_2), where the data is finally consumed. Production and consumption is implemented with uninterpreted procedures `produce(; i)` and `consume(; o)`.

Communication is implemented as a handshake protocol. Variables c_1 and c_2 are records containing three slots. Data is sent in slot $c_i.data$. Slots $c_i.sig$ and $c_i.ack$ are used for synchronisation. If $c_i.sig = c_i.ack$, then the line is empty and data can be sent, else the line is busy and data must be received.

The channel component is implemented as a queue b with maximum capacity N . It either receives data from the producer and stores it in b , or it sends data from the end of the queue to the consumer, depending on the leading await conditions (guards). The channel can be interpreted as a very high level abstraction of a network connection.

Labels l_p and l_c refer to program positions. The property below makes use of these labels and also refers to the internal variables i and o of the producer and the consumer. Therefore these labels and variables are considered output parameters of the two procedures.

```

prodchacon( $N$ ; var  $l_p, i, l_c, o$ )
  var  $c_1 = \text{mkhandshake}(), c_2 = \text{mkhandshake}()$  in
    producer(;  $c_1, i, l_p$ ) || channel( $N$ ;  $c_1, c_2$ ) || consumer(;  $c_2, o, l_c$ )

```



```

producer(var  $c_1, i, l_p$ )
  while true do
    produce(; i);
     $l_p$  : send( $i; c_1$ );

consumer(var  $c_2, o, l_c$ )
  while true do
    receive(;  $c_2, o$ );
     $l_c$  : consume(; o)

channel( $N$ ; var  $c_1, c_2$ )
  var  $b = \text{mkqueue}(N), d = ?$  in
    while true do
      await  $\neg b.\text{full}$ 
         $\wedge c_1.\text{sig} = c_1.\text{ack}$ ;
      receive(;  $c_1, d$ );
       $b := b.\text{enqueue}(d)$ 
    || await  $\neg b.\text{empty}$ 
         $\wedge c_2.\text{sig} \neq c_2.\text{ack}$ ;
       $d := b.\text{next}$ ;
       $b := b.\text{dequeue}$ ;
      send( $d; c_2$ )

send( $d; \text{var } c$ )
  await  $c.\text{sig} = c.\text{ack}$ ;
   $c.\text{data} := d$ ;
   $c.\text{sig} := \neg c.\text{sig}$ 

receive(var  $c, d$ )
  await  $c.\text{sig} \neq c.\text{ack}$ ;
   $d := c.\text{data}$ ;
   $c.\text{ack} := \neg c.\text{sig}$ 

```

Figure 1.1: Example Producer-Channel-Consumer

This chapter examines the following property.

No loss of data: If some data has been produced, then this data will eventually be consumed.

This property can be formulated in temporal logic as follows.

$$\forall D. \square (l'_p \neq l_p \wedge D = i \rightarrow \diamond l'_c \neq l_c \wedge o = D)$$

Always, if the producer has produced data D (producer is currently at label l_p and variable $D = i$), this data is eventually consumed (consumer at label l_c and variable $o = D$).

1.2 Syntax

It is our goal to explicitly support concurrent systems instead of encoding the transition system as a temporal formula. Interactive verification is more intuitive, if systems – which are here defined as a parallel program – are not encoded in a different language.

Our approach differs from Temporal Logic of Actions [18] where concurrent systems are described as temporal formulas in normal form. We rather follow [22] where an Interval Temporal Logic is defined and program constructs are derived as special cases of temporal formulas. As a consequence programs and temporal formulas can be intermixed. This is very important for our modular approach (see Section 1.8). [22] only defines simple program constructs. Here, we shall also support more complex operators as parallel interleaving, blocking and frame assumptions.

As programs and formulas are considered syntactically the same, we refer to both with symbols φ, ψ, \dots . Furthermore, we denote program variables with roman lower case letters c, i, x, \dots and static (or rigid) variables with upper case letters in italic N, D, \dots . In contrast to program variables, the values of static variables are equal in every state. Any type of variable is denoted with italic lower case letters v, w, x, \dots . A letter t refers to an arbitrary term, \vec{t} (resp. \vec{v}) represents a list of terms (resp. a list of program variables). A letter l represents a program variable of boolean type.

1.2.1 Parallel programs

Table 1.1 lists all program constructs with well known constructs of sequential programs on the left – the syntax is similar to the Pascal programming language. Procedures are called with a list of value and var parameters \vec{t} and \vec{x} . On the right hand side, constructs of parallel programs are given. Parallel programs are interleaved, i.e. either a step of the first or a step of the second program is executed. Operator $\varphi_1 \parallel \varphi_2$ nondeterministically chooses an alternative. The sub programs can be guarded with await statements. An alternative can only be chosen, if the leading guards are currently satisfied. These operators originate from [19]. A label marks whether the statement has been entered.

$x := t$	assignment	$\varphi_1 \parallel \varphi_2$	interleaving
$\varphi; \psi$	composition	await ψ	synchronisation
if ψ then φ_1 else φ_2	conditional	$\varphi_1 \sqcap \varphi_2$	choose (Dijkstra)
while ψ do φ	loop	$l : \varphi$	label
var $x = t$ in φ	local variable		
var $x = ?$ in φ	uninitialised local variable		
$p(\vec{t}; \vec{x})$	procedure		

Table 1.1: Constructs of parallel programs

$\square \varphi$	φ holds <i>always</i> from now on in every step
$\diamond \varphi$	φ holds now or <i>eventually</i> later
$\circ \varphi$	there is a next step which satisfies φ (<i>strong next</i>)
$\bullet \varphi$	if there is a next step, it satisfies φ (<i>weak next</i>)
last	<i>last step</i> of execution has been reached
$\varphi; \psi$	first part of execution satisfies φ , second part ψ (<i>chop</i>)
$[\vec{x}]$	only program variables \vec{x} are modified (<i>frame assumption</i>)
blocked	system is <i>blocked</i>

Table 1.2: Operators of temporal formulas

1.2.2 Temporal logic

Besides the standard logical connectives \neg (not), \wedge (and), \vee (or), \rightarrow (implies), and \leftrightarrow (equivalence) we use a number of temporal operators which are explained in Table 1.2. The first group of operators is standard in Linear Temporal Logic, the second group originates from Interval Temporal Logic (ITL [22]). It is important to note that the chop operator $\varphi; \psi$ coincides with sequential composition of programs. In addition, we will support explicit frame assumptions $[\vec{v}]$. A frame assumption corresponds to an infinite formula

$$[\vec{v}] \Leftrightarrow \bigwedge_{w \in \mathbf{Z} \setminus \vec{v}} w' = w$$

which states that all program variables *except* \vec{v} are unchanged. \mathbf{Z} refers to the infinite set of program variables. The operator **blocked** is used to express deadlocks.

First order quantification over static and program variables are also supported, but are not considered in this overview. Operator precedence is as follows.

higher precedence		\longrightarrow		lower precedence		
$\square, \diamond, \circ, \bullet$	$;$	\neg	\wedge	\vee	\rightarrow	\leftrightarrow

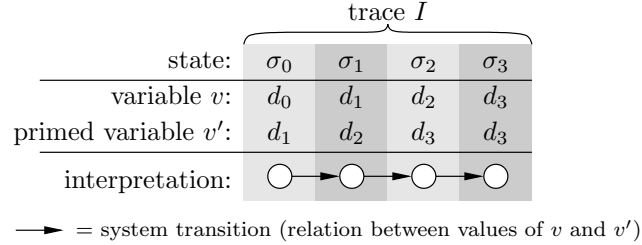


Figure 1.2: Traces of states

1.3 Semantics

The semantics of both formulas and programs can be expressed as a set of runs which are also called traces or intervals. A trace consists of a – finite or infinite – sequence of states. Each state σ assigns values to unprimed variables v and primed variables v' . The value of a primed variable v' is equal to the value of the unprimed variable v in the next state. For example, in Figure 1.2 the value of v' in state σ_1 is d_2 . Transitions between states can be described as relations between v and v' . If there is no next state, the value of v' is equal to v ; the variable stutters. For simplicity, we restrict ourselves to linear sequences of states. Tree-like structures as in Computation Tree Logic (CTL) are not considered here.

The semantics of the standard temporal operators and simple program constructs is taken from Interval Temporal Logic [22] and is omitted here. ITL follows the idea of deriving program constructs from temporal formulas. For example

$$\mathbf{if } \psi \mathbf{ then } \varphi_1 \mathbf{ else } \varphi_2 \quad :\Leftrightarrow \quad \psi \wedge \varphi_1 \vee \neg \psi \wedge \varphi_2$$

However, ITL does not consider more complex program constructs, e.g. interleaving $\varphi_1 \parallel \varphi_2$ and nondeterministic choice $\varphi_1 \sqcap \varphi_2$, simply because their semantics cannot be expressed as an equivalent temporal formula. It is a challenge to define the semantics of these operators not only for programs but for arbitrary sub formulas φ_1, φ_2 . Furthermore, assignments $v := t$ in ITL do not include a frame assumption; all variables which do not change while modifying v must be explicitly mentioned.

1.4 Calculus

It is our goal to define an interactive calculus which allows for intuitive proofs. Interactive proofs in temporal logic are considered difficult. To come up with a calculus which is easy to use, we exploit the idea of symbolic execution here. The strategy of symbolic execution is well known for sequential programs (see Hoare Logic [14] and Dynamic Logic [12, 13]). Executing programs is an intuitive approach and can be automated to a very large

extent. We will apply this strategy also to parallel programs and investigate how far these advantages carry over.

The calculus requires four ingredients. The first three – symbolic execution, induction and a strategy to do modular proofs – are also part of Dynamic Logic. As fourth ingredient a technique – which we call “sequencing” – is required to cope with nondeterministic sequences of steps which are typical for interleaved parallel programs. All four ingredients are explained in the next chapters.

Proof rules are given in a sequent calculus where a sequent

$$\overbrace{\varphi_1, \dots, \varphi_n}^{\text{antecedent}} \vdash \overbrace{\psi_1, \dots, \psi_m}^{\text{succedent}}$$

holds if, and only if, the conjunction of all formulas in the antecedent implies the disjunction of formulas in the succedent.

$$\varphi_1 \wedge \dots \wedge \varphi_n \rightarrow \psi_1 \vee \dots \vee \psi_m$$

Greek letters Γ and Δ abbreviate lists of zero or more formulas. A sequent calculus rule

$$\frac{\overbrace{\Gamma_1 \vdash \Delta_1}^{\text{1st premise}} \quad \dots \quad \overbrace{\Gamma_n \vdash \Delta_n}^{\text{nth premise}}}{\overbrace{\Gamma \vdash \Delta}^{\text{conclusion}}} \text{rule name}$$

ensures that if all of the premises hold then the conclusion is established. In the following, rules are applied backward, i.e., if a proof obligation matches a conclusion, then the rule can be applied to reduce the obligation to a number of “simpler” premises and the proof continues with those.

1.5 Ingredient 1 – Symbolic Execution

1.5.1 Principle

Example Consider the following program.

$$\begin{array}{l} \text{await } c.\text{sig} = c.\text{ack}; \\ c.\text{data} := d; \\ c.\text{sig} := \neg c.\text{sig} \end{array} \quad \parallel \quad \begin{array}{l} \text{await } c.\text{sig} \neq c.\text{ack}; \\ d := c.\text{data}; \\ c.\text{ack} := \neg c.\text{sig} \end{array} \quad (1.1)$$

The program operates on the two variables c and d , with record c containing three slots $c.\text{sig}$, $c.\text{ack}$, and $c.\text{data}$. In order to execute this program, initial values (input) for these variables are required. For example, if initially

$$c.\text{sig} = \text{true} \text{ and } c.\text{ack} = \text{true}$$

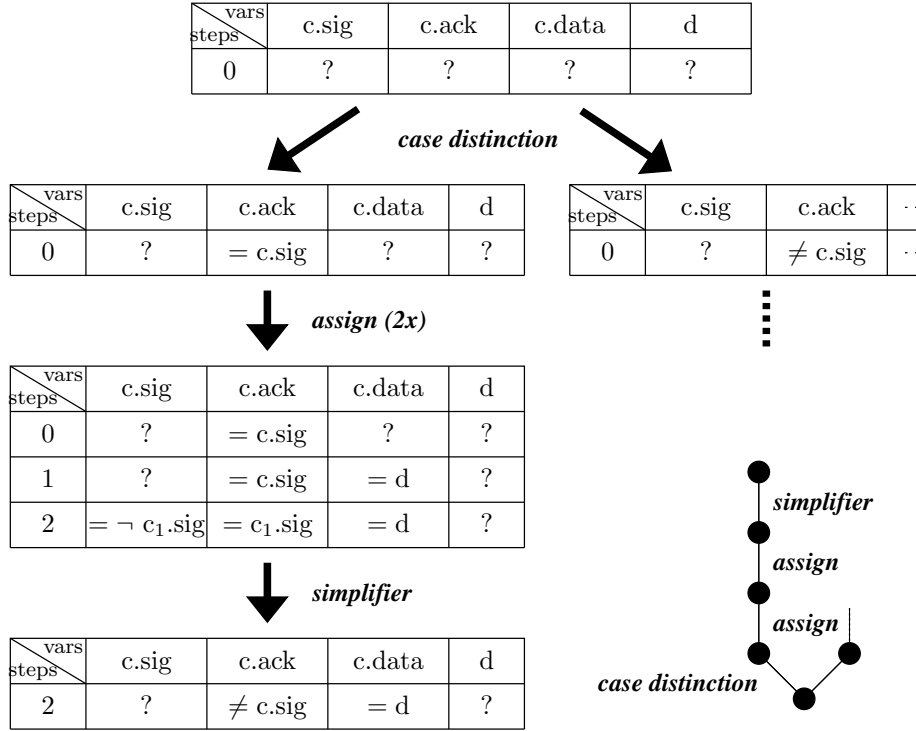


Figure 1.3: Illustration of symbolic execution

then the await condition of the left program is satisfied and the second program is blocked. The left program will be executed first.

Different inputs result in different runs. By providing concrete input to the variables, programs can be tested. But usually there are a large and probably even infinite number of different inputs, and it is not possible to test all cases. This is where symbolic execution can be used. Symbolic execution operates on symbolic variables. Instead of concrete input data, every possible initial state is considered and all of the possible program runs are explored.

Example Figure 1.3 illustrates how to symbolically execute the example program. Initially, all of the variables are unknown. To decide whether the await conditions are fulfilled, we do a case distinction, with the first case assuming $c.ack = c.sig$ and the second case assuming $c.ack \neq c.sig$. In the first case, the await condition of the left program is satisfied and the second program is blocked. We continue to execute the two assignments of the program on the left. First, variable $c.data$ is assigned the value of d , i.e., the value of d is copied into the data slot of record c . Other slots and variables remain untouched. Second, the signal slot of record c is changed, its new value being the negation of the old value – which is the value of the slot in state 1 ($c_1.sig$). Slot $c.sig$ is modified which requires all of the conditions involving

this slot to refer to its old value $c_1.\text{sig}$. In this fashion, all of the statements of the program are considered one after another. This results in more and more complicated conditions for the current values of the variables, and typically a lot of references to old values are introduced. Therefore, it is important to simplify these conditions. The power of simplification can be seen in Figure 1.3. The conditions for the values of the variables in step 2 can be expressed simpler and especially the reference to the old value $c_1.\text{sig}$ can be eliminated.

1.5.2 Executing programs

Example Consider the first process of our example in (1.1). We will use symbolic execution to try to prove that the process implies that eventually $c.\text{data} = d$.

$$\left(\begin{array}{l} \mathbf{await} \ c.\text{sig} = c.\text{ack}; \\ c.\text{data} := d; \\ c.\text{sig} := \neg c.\text{sig} \end{array} \right) \vdash \diamond c.\text{data} = d \quad (1.2)$$

The first program construct is an await statement.

For an await statement **await** ψ , two cases need to be considered. Either the condition ψ holds, or it does not. In the first case execution continues with the rest of the program, in the second case the program is blocked. These two cases can be written down as premises to a calculus rule as follows.

$$\frac{\psi, \varphi, \Gamma \vdash \Delta \quad \neg \psi, [\square], \mathbf{blocked}, \circ (\mathbf{await} \ \psi; \varphi), \Gamma \vdash \Delta}{(\mathbf{await} \ \psi; \varphi), \Gamma \vdash \Delta} \textit{await left}$$

In the second premise, the program is blocked; no variables are changed ($[\square]$) and execution continues with evaluating the condition again in the next state.

Example After applying *await left* to (1.2), the first premise reads

$$c.\text{sig} = c.\text{ack}, \left(\begin{array}{l} c.\text{data} := d; \\ c.\text{sig} := \neg c.\text{sig} \end{array} \right) \vdash \diamond c.\text{data} = d \quad (1.3)$$

The program continues with the two assignments.

An assignment in the antecedent can be executed according to the following rule

$$\frac{v' = t, [\square], \neg \mathbf{blocked}, \circ \varphi, \Gamma \vdash \Delta}{(v := t; \varphi), \Gamma \vdash \Delta} \textit{assign left}$$

The only premise of rule *assign left* states that the value of variable v in the next state is equal to t . Assignments incorporate a frame assumption which ensures that all program variables except the assigned variable are unchanged. Executing an assignment requires exactly one step and in the next state the execution continues with the rest of the program.

Example For (1.3), the application of rule *assign left* leads to the following premise

$$c.\text{sig} = c.\text{ack}, c.\text{data}' = d, [c.\text{data}], \neg \mathbf{blocked}, \circ c.\text{sig} := \neg c.\text{sig} \quad (1.4) \\ \vdash \diamond c.\text{data} = d$$

In this premise, the system is now given as a formula where part of it describes the next transition as a relation between unprimed and primed variables, and the other part gives the system in the next state.

$$\overbrace{c.\text{sig} = c.\text{ack} \wedge c.\text{data}' = d \wedge [c.\text{data}] \wedge \neg \mathbf{blocked}}^{\text{transition}} \wedge \underbrace{\circ c.\text{sig} := \neg c.\text{sig}}_{\text{system in next state}}$$

Arriving at a formula which separates the transition from the system description in the next state is what is here called symbolic execution of a single step. In general, execution of programs gives several premises of the above form, where the premises are the result of case distinctions during the proof. In our example the execution of the await statement gave rise to two separate cases.

The strategy is to execute the different parts of the sequent, i.e., to rewrite the parts to normal form, and to advance a step for the whole sequent afterwards. The program has been rewritten to normal; it remains to also rewrite the temporal formula before a step can be taken for the whole sequent.

1.5.3 Executing temporal formulas

Example The two possible next transitions for the example program in (1.2) have been separated into two premises, one of the premises being (1.4). Before the rest of the program can be executed, the property to verify has to be considered. The example property requires that eventually $c.\text{data} = d$. If d equals $c.\text{data}$ already in the current state, the property is established and the proof is finished. Otherwise the property needs to hold eventually later during execution.

This informal proof idea can be turned into a calculus rule to derive arbitrary eventually properties.

$$\frac{\Gamma \vdash \varphi, \circ \diamond \varphi, \Delta}{\Gamma \vdash \diamond \varphi, \Delta} \textit{eventually right}$$

The only premise requires φ to either hold now or to hold eventually later.

Example Applying this rule to case (1.4) of the example gives

$$\begin{array}{l} c.\text{sig} = c.\text{ack}, c.\text{data}' = d, [c.\text{data}], \neg \mathbf{blocked}, \circ c.\text{sig} := \neg c.\text{sig} \\ \vdash c.\text{data} = d, \circ \diamond c.\text{data} = d \end{array} \quad (1.5)$$

In this formula, not only the next transition of the program has been extracted but also the property has been separated into two parts, the first being concerned with the current state and the second with the rest of the trace. In this fashion, temporal formulas can also be “symbolically executed”.

1.5.4 Advancing a step

Formula (1.5) is considered to be in normal form, i.e., every part either describes the relation between unprimed and primed variables without further reference to the rest of the trace or is prefixed with a next operator. This situation is captured in the following pattern

$$\Gamma_{\mathbf{PL}}(\vec{v}, \vec{v}'), \circ \Gamma \vdash \Delta_{\mathbf{PL}}(\vec{v}, \vec{v}'), \circ \Delta$$

where $\Gamma_{\mathbf{PL}}(\vec{v}, \vec{v}')$ and $\Delta_{\mathbf{PL}}(\vec{v}, \vec{v}')$ are lists of formulas in predicate logic with \vec{v} (resp. \vec{v}') being the list of all unprimed (resp. primed) program variables occurring in $\Gamma_{\mathbf{PL}}$ and $\Delta_{\mathbf{PL}}$. All other formulas Γ and Δ are prefixed with a next operator \circ .

For formulas in normal form, the *step* rule can be applied which is used to advance to the next step in the trace.

$$\frac{\Gamma_{\mathbf{PL}}(\vec{V}_0, \vec{v}), \Gamma \vdash \Delta_{\mathbf{PL}}(\vec{V}_0, \vec{v}), \Delta}{\Gamma_{\mathbf{PL}}(\vec{v}, \vec{v}'), \circ \Gamma \vdash \Delta_{\mathbf{PL}}(\vec{v}, \vec{v}'), \circ \Delta} \textit{step}$$

While the conclusion of this rule considers the complete trace, the premise is relative to the shorter trace starting in the next state. Therefore, all next operators and primes are removed; primed variables in the old state are equal to unprimed variables in the new state. It is important, however, to also remember the values of the unprimed variables in the old state. Therefore fresh, static variables \vec{V}_0 are introduced which replace the unprimed variables \vec{v} .

Example In (1.5) the matter of advancing a step is a bit more complicated, since the formula contains a frame assumption. $\lceil \text{c.data} \rceil$ states that every variable except c.data is unchanged. The frame assumption corresponds to an infinite conjunction of equations. However, in the current state (1.5) only refers to a finite number of dynamic variables c.sig , c.ack , c.data , and d . Only for those variables the frame assumption needs to be expanded and rule *step* can be applied, which leads to the premise

$$\begin{aligned} & C_0.\text{sig} = C_0.\text{ack}, \text{c.data} = D_0, & (1.6) \\ & C_0.\text{sig} = \text{c.sig}, C_0.\text{ack} = \text{c.ack}, D_0 = \text{d}, \quad /* \textit{frame assumption} */ \\ & \neg B_0, \\ & \text{c.sig} := \neg \text{c.sig} \\ & \vdash C_0.\text{data} = D_0, \diamond \text{c.data} = \text{d} \end{aligned}$$

where every unprimed dynamic variable has been replaced by a fresh static variable, and primes and next operators have been removed.

1.5.5 Simplification

Example Stepping has introduced a number of fresh variables and the formula has become rather long. However, it is possible to simplify the PL formulas of (1.6) significantly using

standard techniques: equations can be inserted and eliminated afterwards, resulting in

$$\begin{array}{l} \text{c.sig} = \text{c.ack}, \text{c.data} = \text{d}, \text{c.sig} := \neg \text{c.sig} \\ \vdash \diamond \text{c.data} = \text{d} \end{array} \quad (1.7)$$

The remaining equations cannot be eliminated, as it is in general not possible to replace dynamic variables within temporal operators or programs.

1.5.6 Executing interleaving

So far, the calculus rules to symbolically execute programs and formulas were concerned with one operator each. If the leading operator of a program in the antecedent is an *await* statement, rule *await left* is applied, if it is an assignment, rule *assign left* will do. If there occurs an arbitrary eventually property $\diamond \varphi$ in the succedent, rule *eventually right* matches. Independent of their sub formulas, calculus rules can also be given for conditionals, while loops, always, until and unless operators. However, there is no rule to execute the interleaving operator $\varphi_1 \parallel \varphi_2$ for arbitrary φ_1 and φ_2 .

Consider

$$S \quad \equiv \quad \begin{array}{l} \mathbf{while\ true\ do} \\ \quad i := ? \\ \quad l_1 : \mathbf{skip} \\ \quad \text{send}(i; c) \end{array} \parallel \parallel \begin{array}{l} \mathbf{while\ true\ do} \\ \quad \text{receive}(\ ; c, o) \\ \quad l_2 : \mathbf{skip} \end{array} \quad (1.8)$$

Executing two interleaved programs is to execute a transition from one or the other program and to continue with interleaving the remaining programs. However, the next transitions of the two sub programs are not obvious. They have to be executed first, which involves unwinding the while loop and executing an assignment (for the program on the left) or calling a sub procedure (for the one on the right). Only after the next transitions have been separated from the sub programs, the top level interleaving operator can be executed.

These considerations give rise to a first strategy to execute interleaving.

1. If there is a formula in the antecedent with a top level interleaving operator ...

$$\varphi \parallel \psi, \Gamma \vdash \Delta$$

2. ... symbolically execute its sub formulas first ...

$$(\varphi_{\mathbf{PL}} \wedge \circ \varphi_1) \parallel (\psi_{\mathbf{PL}} \wedge \circ \psi_1), \Gamma \vdash \Delta$$

(where $\varphi_{\mathbf{PL}}$ and $\psi_{\mathbf{PL}}$ are PL formulas without programs or temporal operators)

3. ... and continue with executing the top level interleaving operator using the following rule.

$$\frac{\begin{array}{l} (1) \quad \varphi_{\mathbf{PL}}, \circ (\varphi_1 \parallel (\psi_{\mathbf{PL}} \wedge \circ \psi_1)), \Gamma \vdash \Delta \\ (2) \quad \psi_{\mathbf{PL}}, \circ ((\varphi_{\mathbf{PL}} \wedge \circ \varphi_1) \parallel \psi_1), \Gamma \vdash \Delta \end{array}}{(\varphi_{\mathbf{PL}} \wedge \circ \varphi_1) \parallel (\psi_{\mathbf{PL}} \wedge \circ \psi_1), \Gamma \vdash \Delta} \textit{interleave left}$$

This strategy requires the application of rules not only to the top level operator but to nested sub programs. As it is allowed to interleave not only programs but arbitrary formulas, the nested formulas can be arbitrarily complex and may contain negation, quantification and even more interleaving operators. Designing a calculus to support the interleaving of arbitrary formulas is challenging. In order to apply rules to sub formulas, our approach will be based on rewrite rules rather than sequent calculus rules.

1.6 Ingredient 2 – Sequencing

Particularly for parallel programs, symbolic execution leads to a large number of case distinctions. If two processes are interleaved, then for every step there are two cases: either a transition of the first or of the second process is executed. As a result, the size of the proof tree is exponential in the number of transitions of the interleaved processes. It is, however, very often the case that the order of executing interleaved transitions does not affect the resulting state. Prior analysis of the interleaved processes can be used to reduce the granularity of transitions [3]. However, this approach depends on the property to verify and is only applicable for true parallel programs. Our approach is independent of the property under examination and is compatible to our abstraction technique, i.e., arbitrary temporal formulas can be interleaved.

Example Reconsider program 1.8. Figure 1.4 illustrates the interleaved execution of both processes. For a compact illustration, all operators of the two processes have been labelled and the sub procedures have been expanded as can be seen in the upper part. The tree gives the first part of the the proof. Execution starts at the root at labels l_0, m_0 . In the first step, either a transition of the first or of the second process is executed resulting in two premises satisfying labels l_2, m_0 and l_0, m_3 (execution of **while** and **await** does not require a transition). The next step gives four premises satisfying labels l_4, m_0 , l_2, m_3 , l_2, m_3 , and l_0, m_4 . Note that two of the premises are identical and the sub proofs for both premises will be the same.

In order to avoid duplicate sub proofs, a calculus rule is proposed, which is applied to several conclusions containing the same temporal formulas Γ and Δ .

$$\frac{\bigwedge \Gamma_{\mathbf{PL}}^1 \vee \bigwedge \Gamma_{\mathbf{PL}}^2, \Gamma \vdash \Delta}{\Gamma_{\mathbf{PL}}^1, \Gamma \vdash \Delta \quad \Gamma_{\mathbf{PL}}^2, \Gamma \vdash \Delta} \textit{seq}$$

If the temporal formulas Γ and Δ of two premises are identical, then the same program configuration has been reached in different proof branches. Rule *seq* unites the two

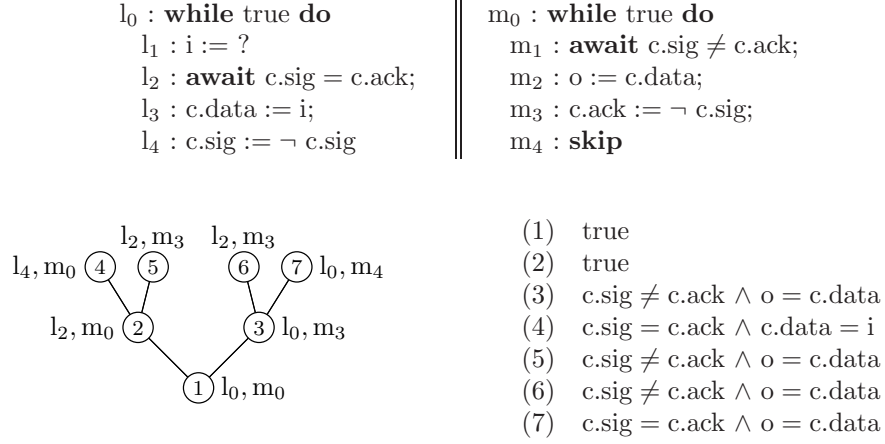


Figure 1.4: Case distinctions for interleaved execution

branches, the PL formulas $\Gamma_{\mathbf{PL}}^1$ and $\Gamma_{\mathbf{PL}}^2$ are combined. It often turns out that the combination of PL formulas can be simplified to a very large extent. If the order of execution did not affect the assignment of variables, then the PL formulas are identical and the disjunction can be eliminated. Only if communication occurs, i.e., the same variables are accessed, the order of steps matters and different symbolic states $\Gamma_{\mathbf{PL}}^1$ and $\Gamma_{\mathbf{PL}}^2$ must be considered.

Example In Figure 1.4, the symbolic states of nodes 5 and 6 are identical. Thus, the two premises can be combined with rule *seq* and the disjunction of PL formulas can be trivially eliminated.

Application of sequent rules to several conclusions results in proof graphs rather than proof trees. We claim, however, that for practical applications, where communication between processes is restricted to a small number of transitions, the size of the proof graph is polynomial in the number of transitions of both processes. Note that rule *seq* is independent of the property to verify. It generally requires that all temporal formulas (parallel programs and properties) of both sequences are identical. Identity of formulas is a syntactic criteria and is subject to automation.

1.7 Ingredient 3 – Induction

Symbolic execution is used to execute a proof obligation step by step. To execute all steps may not be possible, as there could exist infinite traces, or the number of steps may depend on a symbolic value and therefore be arbitrarily large. As an example, reconsider the system description S in (1.8), where the two processes consist of while loops which

never terminate. Consequently, the property

$$S \vdash \square \overbrace{(l'_1 \neq l_1 \wedge D = i \rightarrow \diamond l'_2 \neq l_2 \wedge o = D)}^{P \equiv} \quad (1.9)$$

cannot be proven with symbolic execution only.

In Hoare Logic, loops in sequential programs are proven with an invariant technique. Similarly, in Dynamic Logic induction over the number of cycles can be applied. For parallel programs, this inductive technique can be generalised to a very simple approach which is based on induction over the number of steps. If traces are finite, this boils down to induction over the length of traces. Assuming finite traces, a possible calculus rule is as follows

$$\frac{\bullet \square \mathbf{ih}, \Gamma \vdash \Delta}{\Gamma \vdash \Delta} \text{ ind finite, where } \mathbf{ih} := \bigwedge \Gamma \rightarrow \bigvee \Delta$$

In order to prove $\Gamma \vdash \Delta$ now, a Noetherian induction over the number of steps necessary to reach the final state is performed. As induction hypothesis it is assumed that the property always holds starting in the next state; in other words, after taking a step, the final state is reachable in less steps and the hypothesis is applicable. The weak next operator ensures that, if the last step has already been reached, the induction hypothesis cannot be applied, and the induction is well founded – for finite traces.

For infinite traces, our induction technique makes use of eventually properties $\diamond \varphi$. Induction is over the number of steps necessary to reach the first state satisfying φ .

$$\frac{\Gamma \vdash \diamond \varphi, \Delta \quad (\bullet \mathbf{ih}) \text{ until } \varphi, \Gamma \vdash \Delta}{\Gamma \vdash \Delta} \text{ ind}$$

In a first premise, an eventually property $\diamond \varphi$ is established. In the second premise, the induction hypothesis is applicable in the next state *until* a state is reached, which satisfies φ .

Example Rule *ind* can be used to prove (1.9). As we try to prove a safety condition $\square P$, we can assume the contrary, i.e. $\diamond \neg P$. This establishes the eventually condition required for induction.

$$\frac{S \vdash \diamond \neg P, \square P \quad (\bullet (S \rightarrow \square P)) \text{ until } \neg P, S \vdash \square P}{S \vdash \square P} \text{ ind}$$

While the first premise is easy, the second requires symbolic execution. The while loops of S are unwound and the bodies are executed. In each intermediate step, property P must be assured. Finally, the bodies are completed and program execution is again in front of the while loops. As P holds at least until now, the induction hypothesis is still applicable. Applying the hypothesis leads to

$$S \rightarrow \square P, S, \dots \vdash \square P$$

and the inductive proof is finished.

The example illustrates the basic idea for an inductive proof. However, a complete proof is more complicated, due to the two while loops being interleaved: after initiating induction, the interleaved execution may be such that the two processes are never again in front of the while loops at the same time; consequently, the induction hypothesis is never applicable. Instead of a single induction, the proof requires several nested inductions. It is a challenge to describe a proof strategy such that the induction technique is still easy to use for interleaved programs.

The basic idea to do induction over the number of steps is very simple. Surprisingly, this general idea is suitable for intuitively proving arbitrary temporal properties – even liveness properties – for parallel programs. As will be shown, the same induction technique can be used to also reason in pure temporal logic, or to establish program equivalence. The technique subsumes the invariant rules of Hoare, inductive proofs in Dynamic Logic, and others. Furthermore, it is – in our opinion – easier to use than many existing proof strategies for concurrent systems, e.g., in STeP [20] and TLA [18].

1.8 Ingredient 4 – Abstraction

1.8.1 Goal

With symbolic execution and induction it is possible to investigate arbitrary programs. However, programs can be large and complex and symbolic execution of the whole program can therefore be awkward. To apply the proof strategy to real life applications, a technique to modularise proofs is essential. Our approach is to abstract programs with suitable temporal properties.

Example Reconsider property (1.9) $S \vdash \square P$. As has been sketched in Section 1.7, we initiate induction and afterwards symbolically execute the bodies of the while loops to prove the goal. Here, we assume the sub procedures send and receive to be large and complex. As a consequence, the execution of the complete sub procedures is to be avoided. After unwinding the while loop of the left program and executing the random assignment, we arrive at the following situation.

$$\begin{array}{l} \text{send}(i; c); \\ \mathbf{while\ true\ do} \\ \dots \end{array} \parallel \dots \quad (1.10)$$

(The process on the right may also have been partially executed.) Instead of executing the sub program send, we would like to make use of a lemma

$$\text{send}(D; c) \vdash \overbrace{c.\text{sig} = c.\text{ack}}^A \rightarrow \overbrace{\diamond (c.\text{data} = D \wedge c.\text{sig} \neq c.\text{ack})}^G \quad (1.11)$$

which ensures that under the assumption A , i.e., the channel is currently free, procedure send guarantees G , i.e., eventually data D is written into the channel and the signal is emitted.

$\text{send}(D; c)$	$\text{send}(D; c) \parallel c.\text{data} := E$																																																								
<table border="1" style="border-collapse: collapse; margin: auto;"> <thead> <tr> <th style="width: 20px;"></th> <th style="width: 40px;">c.sig</th> <th style="width: 40px;">c.ack</th> <th style="width: 40px;">c.data</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">0</td> <td style="text-align: center;">?</td> <td style="text-align: center;">= c.sig</td> <td style="text-align: center;">?</td> </tr> <tr> <td style="text-align: center;">1</td> <td style="text-align: center;">?</td> <td style="text-align: center;">= c.sig</td> <td style="text-align: center;">D</td> </tr> <tr> <td style="text-align: center;">2</td> <td style="text-align: center;">?</td> <td style="text-align: center;">\neq c.sig</td> <td style="text-align: center;">D</td> </tr> </tbody> </table>		c.sig	c.ack	c.data	0	?	= c.sig	?	1	?	= c.sig	D	2	?	\neq c.sig	D	<table border="1" style="border-collapse: collapse; margin: auto;"> <thead> <tr> <th style="width: 20px;"></th> <th style="width: 40px;">c.sig</th> <th style="width: 40px;">c.ack</th> <th style="width: 40px;">c.data</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">0</td> <td style="text-align: center;">?</td> <td style="text-align: center;">= c.sig</td> <td style="text-align: center;">?</td> </tr> <tr> <td style="text-align: center;">1</td> <td style="text-align: center;">?</td> <td style="text-align: center;">= c.sig</td> <td style="text-align: center;">D</td> </tr> <tr> <td style="text-align: center;">2</td> <td style="text-align: center;">?</td> <td style="text-align: center;">= c.sig</td> <td style="text-align: center;">E</td> </tr> <tr> <td style="text-align: center;">3</td> <td style="text-align: center;">?</td> <td style="text-align: center;">\neq c.sig</td> <td style="text-align: center;">E</td> </tr> </tbody> </table>		c.sig	c.ack	c.data	0	?	= c.sig	?	1	?	= c.sig	D	2	?	= c.sig	E	3	?	\neq c.sig	E																				
	c.sig	c.ack	c.data																																																						
0	?	= c.sig	?																																																						
1	?	= c.sig	D																																																						
2	?	\neq c.sig	D																																																						
	c.sig	c.ack	c.data																																																						
0	?	= c.sig	?																																																						
1	?	= c.sig	D																																																						
2	?	= c.sig	E																																																						
3	?	\neq c.sig	E																																																						
$\diamond (c.\text{sig} \neq c.\text{ack} \wedge c.\text{data} = D) \parallel c.\text{data} := E$																																																									
<table border="1" style="border-collapse: collapse; margin: auto;"> <thead> <tr> <th style="width: 20px;"></th> <th style="width: 40px;">c.sig</th> <th style="width: 40px;">c.ack</th> <th style="width: 40px;">c.data</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">0</td> <td style="text-align: center;">?</td> <td style="text-align: center;">= c.sig</td> <td style="text-align: center;">?</td> </tr> <tr> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> </tr> <tr> <td style="text-align: center;">n</td> <td style="text-align: center;">?</td> <td style="text-align: center;">= c.sig</td> <td style="text-align: center;">E</td> </tr> <tr> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> </tr> <tr> <td style="text-align: center;">m</td> <td style="text-align: center;">?</td> <td style="text-align: center;">\neq c.sig</td> <td style="text-align: center;">D</td> </tr> <tr> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> </tr> </tbody> </table>		c.sig	c.ack	c.data	0	?	= c.sig	?	⋮	⋮	⋮	⋮	n	?	= c.sig	E	⋮	⋮	⋮	⋮	m	?	\neq c.sig	D	⋮	⋮	⋮	⋮	<table border="1" style="border-collapse: collapse; margin: auto;"> <thead> <tr> <th style="width: 20px;"></th> <th style="width: 40px;">c.sig</th> <th style="width: 40px;">c.ack</th> <th style="width: 40px;">c.data</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">0</td> <td style="text-align: center;">?</td> <td style="text-align: center;">= c.sig</td> <td style="text-align: center;">?</td> </tr> <tr> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> </tr> <tr> <td style="text-align: center;">n</td> <td style="text-align: center;">?</td> <td style="text-align: center;">\neq c.sig</td> <td style="text-align: center;">D</td> </tr> <tr> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> </tr> <tr> <td style="text-align: center;">m</td> <td style="text-align: center;">?</td> <td style="text-align: center;">\neq c.sig</td> <td style="text-align: center;">E</td> </tr> <tr> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> <td style="text-align: center;">⋮</td> </tr> </tbody> </table>		c.sig	c.ack	c.data	0	?	= c.sig	?	⋮	⋮	⋮	⋮	n	?	\neq c.sig	D	⋮	⋮	⋮	⋮	m	?	\neq c.sig	E	⋮	⋮	⋮	⋮
	c.sig	c.ack	c.data																																																						
0	?	= c.sig	?																																																						
⋮	⋮	⋮	⋮																																																						
n	?	= c.sig	E																																																						
⋮	⋮	⋮	⋮																																																						
m	?	\neq c.sig	D																																																						
⋮	⋮	⋮	⋮																																																						
	c.sig	c.ack	c.data																																																						
0	?	= c.sig	?																																																						
⋮	⋮	⋮	⋮																																																						
n	?	\neq c.sig	D																																																						
⋮	⋮	⋮	⋮																																																						
m	?	\neq c.sig	E																																																						
⋮	⋮	⋮	⋮																																																						

Figure 1.5: Comparison of traces for lemma application

Once a temporal property has been proven for a procedure, the property should be usable to establish other properties. When proving other properties for the same procedure, the proposed strategy is to replace the procedure by an already established (simpler) property and to symbolically execute the property instead of the complex procedure. Simply replacing sub programs with formulas is possible, because the semantics of formulas and programs is defined such that they can be mixed (see Sect. 1.3). As a consequence, abstraction is similar to standard lemma application.

Example Replacing `send` in (1.10) leads to

$$\left(\begin{array}{l}
 c.\text{sig} = c.\text{ack} \\
 \rightarrow \diamond (c.\text{data} = i \wedge c.\text{sig} \neq c.\text{ack}); \\
 \mathbf{while\ true\ do} \\
 \quad \dots
 \end{array} \right) \parallel \dots$$

1.8.2 Problem

Example Lemma (1.11) is easy to verify. Because of A , the `await` condition of `send` is immediately satisfied and after two assignments, the guarantee G is established. Figure 1.5 illustrates in the upper left the resulting trace. In step 2, the data slot of channel c is D and

the signal flag has been inverted; the guarantee is established. However, if the program is executed in parallel to another program which also manipulates record c , e.g.

$$(\text{send}(D; c) \parallel c.\text{data} := E) \stackrel{?}{\vdash} A \rightarrow G \quad (1.12)$$

does the property still hold? In the upper right of Figure 1.5 one possible trace of the interleaving is displayed. The two assignments of send are interleaved with the execution of the assignment running in parallel (step 2). On this trace, there is no state with both $c.\text{data} = D$ and $c.\text{sig} \neq c.\text{ack}$; property (1.12) does not hold! But, if lemma (1.11) is applied and send is replaced

$$((A \rightarrow G) \parallel c.\text{data} := E) \stackrel{?}{\vdash} A \rightarrow G$$

then the possible traces are different. The guarantee G states for the process on the left that eventually there is a state satisfying both $c.\text{data} = D$ and $c.\text{sig} \neq c.\text{ack}$. If G is interleaved with the assignment, two types of traces are possible which are depicted in the lower part of Figure 1.5. On the left, $c.\text{data} := E$ is executed before the first process reaches the state of interest, on the right the assignment is executed later. In both cases, property $A \rightarrow G$ is satisfied.

The example illustrates a major problem of the suggested approach. It is wrong to simply replace a program with a property which holds for the program alone but is not valid if the program runs in parallel to other programs.

1.8.3 Idea

A modular approach which avoids the illustrated problem above is TLA [18]. A TLA formula modelling a system allows for so called *stuttering steps* which represent steps of the environment. TLA formulas adhere to a very restricted normal form as follows:

$$\underbrace{\text{Init}}_{\text{initial state}} \wedge \square \underbrace{[\mathcal{A}_1 \vee \dots \vee \mathcal{A}_n]}_{\text{actions}} \underbrace{\langle v_1, \dots, v_m \rangle}_{\text{stuttering step}} \wedge \underbrace{\mathcal{F}}_{\text{fairness assumptions}}$$

where Init is a predicate on the initial state (e.g. $x = 0$), actions $\mathcal{A}_1, \dots, \mathcal{A}_n$ model the possible transitions as relation between unprimed and primed variables (e.g. $x' = x + 1$), and the fairness assumptions \mathcal{F} are very restricted temporal formulas (e.g., $\diamond \square \text{enabled}(\mathcal{A}) \rightarrow \square \diamond \text{exec}(\mathcal{A})$). The semantics of the stuttering step is as follows:

$$\square \left[\bigvee_i \mathcal{A}_i \right]_{\langle v_1, \dots, v_m \rangle} \quad :\equiv \quad \square \left(\bigvee_i \mathcal{A}_i \vee (v'_1 = v_1 \wedge \dots \wedge v'_m = v_m) \right)$$

The system always executes an action *or* it stutters, i.e., selected variables v_1, \dots, v_m are unchanged and all the other variables may change arbitrarily. Stuttering models an environment step.

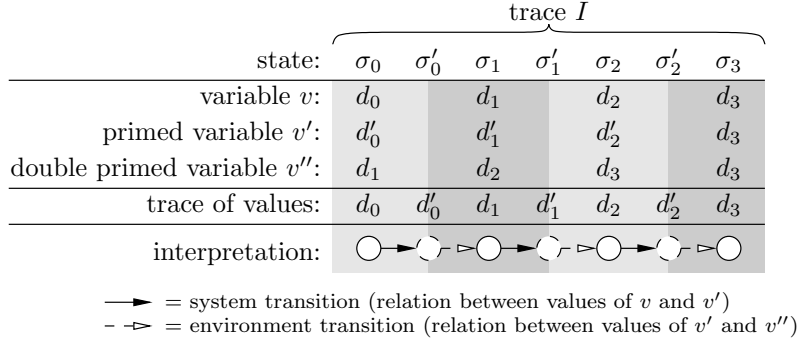


Figure 1.6: Semantics considers environment transitions

In TLA, the environment is considered part of the system description. If properties are proven for a system described in normal form, this property holds for arbitrary environments, except that the environment is required not to modify certain variables. As a consequence, large systems can be composed from smaller modules as described in [1]. However, the proposed normal form of TLA formulas is rather inconvenient. To arrive at a more intuitive solution, we propose to consider the environment already as part of the semantics. This is described next.

1.8.4 Semantics (Version 2)

In Section 1.3, transitions are described as a relation between unprimed and primed variables. Formulas referring to v and v' restrict both the system and environment steps, the steps cannot be distinguished. In the following, it is claimed that this is the cause for the problem above. To enable modular system descriptions, the idea is to introduce *double primed variables* v'' as depicted in Figure 1.6. The relation between v and v' defines the system transition and the relation between v' and v'' is interpreted as an environment step. The transition from state σ_0 to σ_1 consists of a system transition to arrive at an intermediate state σ'_0 and is followed by an environment transition.

The major advantage of separating system and environment transitions in the semantics is that an arbitrary environment is considered without stipulating syntactic restrictions for the formula describing the system behaviour: the system is described with arbitrary formulas referring to v and v' . In addition to the system behaviour, environment assumptions can be expressed as formulas referring to v' and v'' . The assumptions can be more complex than simple stuttering of variables as in TLA.

Example Lemma 1.11 is not valid with respect to the extended semantics. The environment can arbitrarily change slot $c.data$ between the execution of the first and the second assignment of procedure `send`. Consequently, an additional assumption is necessary to prevent the

environment from doing so.

$$\begin{array}{l} \text{send}(D; c) \\ \vdash \quad c.\text{sig} = c.\text{ack} \wedge \square (c'.\text{sig} = c'.\text{ack} \rightarrow c'' = c') \\ \rightarrow \diamond c'.\text{data} = D \wedge c'.\text{sig} \neq c'.\text{ack} \end{array}$$

While the signal flag still equals the acknowledge flag, the environment is not allowed to modify the channel. This assumption is not fulfilled by the program $c.\text{data} := E$, which is running in parallel in (1.12). As a consequence, the lemma is useless to prove (1.12) and the problem of proving false properties is avoided.

The example illustrates the use of double primed variables. It is a challenge, though, to define the semantics of all operators and to give calculus rules such that everything is compatible with the extended semantics. *It turns out that the introduction of double primed variables allows for a compositional semantics of all operators including the interleaving operator.*

1.9 Outlook

The main goals of this work are as follows.

- Definition of a proof strategy based on symbolic execution with induction for the interactive verification of temporal properties for concurrent systems. Symbolic execution promises to make interactive proofs in temporal logic more intuitive and more automatic.
- Definition of a simple induction technique which is general enough to prove all kinds of properties including safety and liveness properties. This is to avoid different proof strategies for different properties.
- Definition of a compositional semantics for all operators including interleaving of parallel processes. This is to modularise proofs. Our approach is to introduce double primed variables into the semantics.

In order to achieve these goals, the following challenges must be met.

- For the abstraction of sub programs with temporal properties, a common semantics for temporal formulas and parallel programs including interleaving, nondeterministic choice and explicit blocking is required.
- Assignments with frame assumptions should be supported. This is to avoid explicit reference in every transition of variables which stutter. This is also a prerequisite for our abstraction technique.
- If arbitrary temporal formulas are interleaved, symbolic execution requires the application of rules to sub formulas. Our approach is to use rewrite rules instead of sequent rules in order to rewrite sub formulas to normal form.

- Exponential proof size must be avoided. Our approach is to use sequencing which is independent of the property to verify.

Chapter 2

Syntax and Semantics

The goal of this chapter is to define a logic where system behaviour can be described as parallel programs and properties are formulated in Linear Temporal Logic (LTL). The syntax of Section 2.1 is close to [19], the semantics of Section 2.2, however, is more in the style of Interval Temporal Logic (ITL) [22] where programs and formulas can be mixed. Different from ITL, frame assumptions and interleaving are also defined. Double primed variables are introduced to receive a compositional semantics for interleaving. As a consequence, systems are open by default, the environment is explicitly considered on the semantics level. The idea of compositionality is inspired by [18], but our approach promises to give more intuitive system descriptions.

2.1 Syntax

We will define a sorted first order expression logic with static and dynamic variables, and operations (including functions and predicates). Furthermore, procedures modularise programs. Altogether these elements define a signature.

Definition 1 (*Signature SIG*) A signature $\mathbf{SIG} = (\mathbf{S}, \mathbf{OP}, \mathbf{PROC}, \mathbf{X}, \mathbf{Y}, \mathbf{Z})$ consists of

- a finite set of sorts \mathbf{S} ,
- a finite family \mathbf{OP} of operations $\mathbf{OP}_{s_1, \dots, s_n}^s$ with argument sorts s_1, \dots, s_n and target sort s

$$\mathbf{OP} := \bigcup_{s_i, s \in \mathbf{S}} \mathbf{OP}_{s_1, \dots, s_n}^s,$$

- a finite family **PROC** of procedures $\mathbf{PROC}_{s_1, \dots, s_n}^{s_1^r, \dots, s_j^r}$ with value parameter sorts s_1, \dots, s_n and reference parameter sorts s_1^r, \dots, s_j^r

$$\mathbf{PROC} := \bigcup_{s_i, s_j^r \in \mathbf{S}} \mathbf{PROC}_{s_1, \dots, s_n}^{s_1^r, \dots, s_j^r} ,$$

- and families **X**, **Y** and **Z** of countably infinite sets of static variables \mathbf{X}_s , dynamic variables \mathbf{Y}_s , and program variables \mathbf{Z}_s

$$\mathbf{X} := \bigcup_{s \in \mathbf{S}} \mathbf{X}_s , \quad \mathbf{Y} := \bigcup_{s \in \mathbf{S}} \mathbf{Y}_s , \quad \mathbf{Z} := \bigcup_{s \in \mathbf{S}} \mathbf{Z}_s .$$

Every signature is assumed to include data types for booleans and natural numbers $\{\text{bool}, \text{nat}\} \subseteq \mathbf{S}$ at the least. The set of operations always includes predicates

$$\{\text{true}, \text{false}, \neg, \wedge, \vee, \rightarrow, \leftrightarrow\} \subseteq \mathbf{OP}_{\text{bool}} ,$$

and functions

$$\{0, \text{succ}, +\} \subseteq \mathbf{OP}_{\text{nat}} .$$

A special program variable of boolean type $\text{blk} \in \mathbf{Z}_{\text{bool}}$ is used to mark whether a parallel program is blocked.

The syntax of all logical operators is defined next. The logic does not distinguish between formulas, terms, or even parallel programs; every operator defines an expression. This ensures maximum flexibility to mix programs with formulas, especially allowing system descriptions to be abstracted by temporal properties.

Definition 2 (*Syntax of expressions \mathbf{E}*) For a given signature **SIG**, the set of expressions **E** is a family of sorted expressions \mathbf{E}_s with

$$\mathbf{E} := \bigcup_{s \in \mathbf{S}} \mathbf{E}_s ,$$

where for all $s \in \mathbf{S}$, \mathbf{E}_s are defined to be the smallest sets satisfying the following:

- (variables) if $X \in \mathbf{X}_s$ and $w \in \mathbf{Y}_s \cup \mathbf{Z}_s$, then
 - $X \in \mathbf{E}_s$ (static variable),
 - $w \in \mathbf{E}_s$ (unprimed dynamic variable),
 - $w' \in \mathbf{E}_s$ (primed dynamic variable), and
 - $w'' \in \mathbf{E}_s$ (double primed dynamic variable),
- if $f \in \mathbf{OP}_{s_1, \dots, s_n}^s$ and $e_i \in \mathbf{E}_{s_i}$, then

- $f(e_1, \dots, e_n) \in \mathbf{E}_s$ (function call),
- if $e_1, e_2 \in \mathbf{E}_s$, then
 - $e_1 = e_2 \in \mathbf{E}_{\text{bool}}$ (equation),
- if $\varphi \in \mathbf{E}_{\text{bool}}$, $v \in \mathbf{X} \cup \mathbf{Y}$ and $x \in \mathbf{Z} \setminus \{\text{blk}\}$, then
 - $\exists v. \varphi \in \mathbf{E}_{\text{bool}}$ (existential quantification),
 - $\exists x. \varphi \in \mathbf{E}_{\text{bool}}$ (hiding).
- (ITL operators) if $\varphi, \psi \in \mathbf{E}_{\text{bool}}$, then
 - $\varphi; \psi \in \mathbf{E}_{\text{bool}}$ (chop),
 - $\varphi^* \in \mathbf{E}_{\text{bool}}$ (star), and
 - **step** $\in \mathbf{E}_{\text{bool}}$,
- (LTL operators) if $\varphi, \psi \in \mathbf{E}_{\text{bool}}$, then
 - φ **until** $\psi \in \mathbf{E}_{\text{bool}}$,
- (sequential programs) if $x_i \in \mathbf{Z} \setminus \{\text{blk}\}$, then
 - $[x_1, \dots, x_n] \in \mathbf{E}_{\text{bool}}$ (frame assumption),
- (parallel programs) if $l_1, \varphi, l_2, \psi \in \mathbf{E}_{\text{bool}}$, then
 - $l_1 :: \varphi \parallel^< l_2 :: \psi \in \mathbf{E}_{\text{bool}}$ (left interleaving),
 - $\varphi \parallel \psi \in \mathbf{E}_{\text{bool}}$ (nondeterministic choice),
 - $\{\varphi\} \in \mathbf{E}_{\text{bool}}$ (atomic step),
- (procedures) if $\text{proc} \in \mathbf{PROC}_{s_1^r, \dots, s_m^r}^{s_1^r, \dots, s_m^r}$, $e_i \in \mathbf{E}_{s_i}$, and $x_j \in \mathbf{Z}_{s_j^r}$ mutually different ($x_i \neq x_j$), then
 - $\text{proc}(e_1, \dots, e_n; x_1, \dots, x_m) \in \mathbf{E}_{\text{bool}}$ (procedure call),
- (system operators) if $\varphi, \psi \in \mathbf{E}_{\text{bool}}$, then
 - $\langle \varphi \rangle \psi \in \mathbf{E}_{\text{bool}}$ (diamond).

The logic includes operators from Interval Temporal Logic (ITL) and Linear Temporal Logic (LTL). For parallel programs, operators for interleaving and blocking expressions as well as Dijkstra's nondeterministic choice are included as basic operators. Furthermore, an operator for an explicit frame assumption is defined. All the other typical program operators will be derived as abbreviations below. In this context, it is important to note that sequential composition of programs coincides with the chop operator of ITL. Our temporal logic is linear – in contrast to branching time logics; however, the diamond

operator allows existential quantification over traces in a restricted manner. Further details on system operators will be discussed in Section 2.2.10.

The basic left interleaving operator $l_1 :: \varphi \parallel^< l_2 :: \psi$ interleaves two “processes” φ and ψ and gives precedence to the left process, i.e. a transition of φ is executed first. The single purpose of the additional formulas l_1 and l_2 is to mark whether the execution of one of the two processes has been deferred. The “markers” are important for our induction technique in Sect. 6. A default marker is false and is often omitted in the following; we simply write $\varphi \parallel^< \psi$ to denote $\text{false} :: \varphi \parallel^< \text{false} :: \psi$.

In the following, \mathbf{F} abbreviates the set of boolean expressions \mathbf{E}_{bool} . Greek letters $\varphi, \psi, \chi \in \mathbf{F}$ always denote formulas while variables with upper case letters in italic $X \in \mathbf{X}$ represent static, variables with lower case letters in italic $x \in \mathbf{Y}$ dynamic, and variables with roman lower case letters $x \in \mathbf{Z}$ program variables – if not stated otherwise.

Additional common logical operators will be defined as abbreviations. The complete list of additional operators is as follows.

Definition 3 (*Abbreviations*)

- (*Derived quantifiers*) Let $v \in \mathbf{X} \cup \mathbf{Y}$.
 - $\forall v. \varphi \in \mathbf{F}$ (*universal quantification*),
 - $\forall x. \varphi \in \mathbf{F}$ (*universal hiding*), and
- (*Derived ITL operators*)
 - **more**, **last**, **inf**, **finite** $\in \mathbf{F}$, and
 - **finally** $\varphi \in \mathbf{F}$.
- (*Derived LTL operators*)
 - $\square \varphi \in \mathbf{F}$ (*always*),
 - $\blacksquare \varphi \in \mathbf{F}$ (*weak always*),
 - $\diamond \varphi \in \mathbf{F}$ (*eventually*),
 - φ **unless** $\psi \in \mathbf{F}$,
 - $\circ \varphi \in \mathbf{F}$ (*strong next*), and
 - $\bullet \varphi \in \mathbf{F}$ (*weak next*).
- (*Derived sequential programs*) Let $x \in \mathbf{Z}_s$, and $e \in \mathbf{E}_s$.
 - $x := e \in \mathbf{F}$ (*assignment*),
 - $x := ? \in \mathbf{F}$ (*random assignment*),
 - **skip** $\in \mathbf{F}$ (*no operation*),
 - **if** ψ **then** φ_1 **else** $\varphi_2 \in \mathbf{F}$ (*conditional*),

- **if** ψ **then** $\varphi \in \mathbf{F}$
- **while** ψ **do** $\varphi \in \mathbf{F}$ (*loop*),
- **var** $x = e$ **in** $\varphi \in \mathbf{F}$ (*initialised local variable*),
- **var** $x = ?$ **in** $\varphi \in \mathbf{F}$ (*uninitialised local variable*), and
- **abort** $\in \mathbf{F}$ (*non-terminating program*).
- (*Derived parallel programs*) Let $l_1, l_2 \in \mathbf{F}$, $l \in \mathbf{Z}_{\text{bool}}$.
 - **await** $\varphi \in \mathbf{F}$ (*synchronisation*),
 - **blocked** $\in \mathbf{F}$ (*blocking*), and
 - $l_1 :: \varphi \parallel l_2 :: \psi \in \mathbf{E}_{\text{bool}}$ (*interleaving*),
 - $l_1 :: \varphi \parallel^> l_2 :: \psi \in \mathbf{E}_{\text{bool}}$ (*right interleaving*),
 - $l_1 :: \varphi \parallel_b^< l_2 :: \psi \in \mathbf{E}_{\text{bool}}$ (*blocked left interleaving*),
 - $l_1 :: \varphi \parallel_b^> l_2 :: \psi \in \mathbf{E}_{\text{bool}}$ (*blocked right interleaving*),
 - $\varphi \parallel^b \psi \in \mathbf{E}_{\text{bool}}$ (*blocked nondeterministic choice*),
 - $l : \varphi \in \mathbf{F}$ (*label*).
- (*Derived system operators*)
 - $[\varphi] \psi \in \mathbf{F}$ (*box*).

The operator precedence is defined as in Table 2.1. If formulas stretch over several lines, then we often omit brackets, provided that they follow from indentation.

Example (Operator precedence) Formula

$$\square \varphi \text{ until true} \wedge \circ \text{ false} \rightarrow \varphi \wedge \text{ true}$$

is parsed as if the following brackets were used:

$$(((\square \varphi) \text{ until true}) \wedge (\circ \text{ false})) \rightarrow (\varphi \wedge \text{ true})$$

The program operators of our logic are such that (interleaved) parallel programs can be notated in a natural style. As an example, we have taken a parallel program from [19] to compute the binomial coefficient.

Example (Binomial coefficient) The procedure $\text{Binom}(n, k; b)$ with value parameters n, k

operator precedence		
highest	=	
	:=	
	*	
	□, ■, ◇, ○, ●, :	
	until, unless	
	if then else, while do, var in, await	
	, ^{<} , ^{>} , _b ^{<} , _b ^{>} , []	
	;	
	< >, []	
	¬	
	∧	
	∨	
	→	
	↔	
	lowest	∀, ∃, ∀, ∃

Table 2.1: Operator precedence

and reference parameter b is defined to be equivalent to

$$\text{Binom}(n, k; b)$$

```

↔ var n = n, k = k in
  var y1 = n, y2 = 1, t1 = ?, t2 = ?, r = 1 in
    b := 1;
    while y1 > n - k do || while y2 ≤ k do do
      P(; r);                await y1 + y2 ≤ n;
      t1 := b * y1;          P(; r);
      b := t1;               t2 := b div y2;
      V(; r);                b := t2;
      y1 := y1 - 1;          V(; r);
                           y2 := y2 + 1
  
```

The parallel program calculates the binomial coefficient

$$b = \binom{n}{k} = \frac{n!}{k! * (n - k)!} = \frac{n * (n - 1) * \dots * (n - k + 1)}{1 * 2 * \dots * k}$$

using two parallel processes. The first process takes care of the numerator, the second process of the denominator. Separate assignments are used for read and write access to variable b , the intermediate result being stored in auxiliary variables t_1 and t_2 . Access to the shared variable b is synchronised with a semaphore r . An additional guard **await** $y_1 + y_2 \leq n$ ensures that the remainder of the divisions in the second process is always 0.

2.2 Semantics

As usual, algebras are used to define the semantics of sorts and operations. An algebra interprets the sorts and operations of a given signature. Here, an algebra also contains a relation for each procedure of a given signature **SIG**.

Definition 4 (*Algebra \mathcal{A}*) Given a signature $\mathbf{SIG} = (\mathbf{S}, \mathbf{OP}, \mathbf{PROC}, \mathbf{X}, \mathbf{Y}, \mathbf{Z})$, an algebra $\mathcal{A}_{\mathbf{SIG}}$ consists of

- a family \mathbf{D} (called domain) of nonempty sets \mathbf{D}_s for each $s \in \mathbf{S}$ with

$$\mathbf{D} := \bigcup_{s \in \mathbf{S}} \mathbf{D}_s,$$

- a function

$$f_{\mathcal{A}} : \mathbf{D}_{s_1} \times \dots \times \mathbf{D}_{s_n} \rightarrow \mathbf{D}_s$$

for each operation $f \in \mathbf{OP}_{s_1, \dots, s_n}^s$, and

- a relation

$$proc_{\mathcal{A}} : \mathbf{D}_{s_1} \times \dots \times \mathbf{D}_{s_n} \times \left(\mathbf{D}_{s_1}^{(i)} \times \dots \times \mathbf{D}_{s_m}^{(i)} \times \mathbf{D}_{\text{bool}}^{(i)} \right)_i$$

for each procedure $proc \in \mathbf{PROC}_{s_1, \dots, s_n}^{s_1^r, \dots, s_m^r}$.

With \mathbf{A} we refer to the set of all algebras.

The relation defining the semantics of a procedure deserves further explanation. First, the relation takes the initial values of the value parameters. Second, it receives a (finite or infinite) sequence of values \mathbf{D}^{m+1} for each reference parameter. Variable blk is an implicit reference parameter for each procedure. Further details of, and an example for, the semantics of procedures follow in Sect. 2.2.9 below.

For every algebra \mathcal{A} , the domain \mathbf{D}_{bool} is assumed to consist of the two truth values of boolean logic tt and ff , and the domain \mathbf{D}_{nat} to coincide with the set of natural numbers \mathbb{N} .

Algebras are used to evaluate operators and procedures. In order to evaluate the variables of a given signature **SIG**, we define the notion of states.

Definition 5 (*State σ*) Given a signature **SIG** and an algebra $\mathcal{A}_{\mathbf{SIG}}$, a state σ consists of functions

$$\sigma_s : \mathbf{X}_s \cup \mathbf{Y}_s \cup \mathbf{Z}_s \mapsto \mathbf{D}_s$$

for each sort $s \in \mathbf{S}$ mapping static, dynamic, and program variables to domain elements. A state σ is also called a valuation.

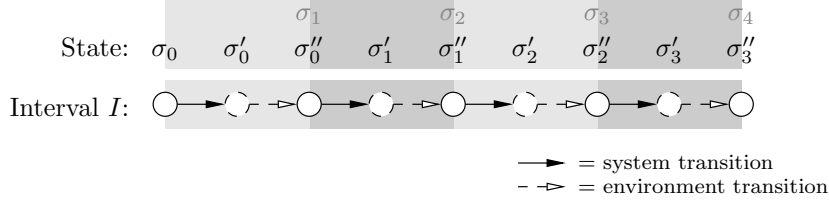


Figure 2.1: An interval as sequence of states

The symbol Σ refers to the set of all states.

For temporal logic, a single state does not suffice. Temporal formulas are rather evaluated on a sequence of states (σ_0, \dots) . The notion of sequences (or intervals) of states is central to this work and is therefore defined in the following separate section.

2.2.1 Intervals

We evaluate temporal formulas on linear sequences of states – often called traces. Similar to Interval Temporal Logic, we explicitly consider finite and infinite traces and therefore also refer to a sequence of states as an interval. To ensure modularity, we define intervals to allow for separate environment transitions as has been discussed in the overview Section 1.8. For intuition, compare the following formal definition of intervals I to Figure 2.1.

Definition 6 (Interval I) Let $\bar{n} \in \mathbb{N}^\infty$. An interval

$$I = (\sigma_0, \sigma'_0, \sigma''_0, \dots, \sigma'_{\bar{n}-1}, \sigma''_{\bar{n}-1})$$

consists of

- an initial state σ_0 , and
- a finite or infinite and possibly empty sequence of transitions

$$(\sigma'_i, \sigma''_i)_{i=0}^{\bar{n}-1},$$

where $\sigma'_i(X) = \sigma''_i(X) = \sigma_0(X)$ for all $i < \bar{n}$, $X \in \mathbf{X}$.

$|I| := \bar{n}$ is defined as the length of an interval I . The interval is called empty if $|I| = 0$, it is called infinite if $|I| = \infty$. Unprimed states σ_{i+1} are defined to be equal to double primed states σ''_i . The relation between σ_i and σ'_i is called a system transition, the relation between σ'_i and σ''_i is interpreted as an environment transition.

The symbol \mathbf{I} refers to the set of all intervals.

The first state σ_0 of an interval contains the initial values for each variable. In the second state σ'_0 the values of the variables after the first system transition are stored. The next state σ''_0 reflects the state after the first environment transition. This double primed state is also referred to as σ_1 . In this manner, system and environment transitions alternate. Note that in an interval only the dynamic and program variables change while static variables X are assigned to the same values $\sigma_0(X)$ by definition. Also note that an empty interval contains the initial state σ_0 .

A selection of additional functions concerning intervals is helpful in the following sections and is defined next.

Definition 7 (*Auxiliary definitions for intervals*) Let $w \in \mathbf{Y} \cup \mathbf{Z}$. Let I be a finite or infinite interval (σ_0, \dots) and $\bar{n} = |I|$. Then the following auxiliary functions on intervals are defined.

$$\begin{aligned}
I(i) &:= \begin{cases} \sigma_i, & \text{if } i \leq \bar{n} \\ \sigma_{\bar{n}}, & \text{otherwise} \end{cases} \\
I(i)' &:= \begin{cases} \sigma'_i, & \text{if } i < \bar{n} \\ \sigma_{\bar{n}}, & \text{otherwise} \end{cases} \\
I(i)'' &:= I(i+1) \\
I|_i &:= \begin{cases} (\sigma_i, \dots), & \text{if } i \leq \bar{n} \\ (\sigma_{\bar{n}}), & \text{otherwise} \end{cases} \\
I^i &:= \begin{cases} (\sigma_0, \dots, \sigma_i), & \text{if } i \leq \bar{n} \\ (\sigma_0, \dots, \sigma_{\bar{n}}), & \text{otherwise} \end{cases} \\
I|_i^j &:= (I^j)|_i \\
I(X) &:= I(0)(X) \\
I(w) &:= (I(0)(w), I(0)'(w), I(1)(w), \dots)
\end{aligned}$$

$I(i)$ selects the i th state of an interval. $I|_i$ (resp. I^i) is a *postfix* (resp. *prefix*), and $I|_i^j$ a *subinterval* of I . Because static variables do not change, the value of X in an interval can be directly accessed with $I(X)$. For dynamic and program variables, $I(w)$ evaluates to a sequence of values.

Definition 8 (*Semantics*) The following function

$$\llbracket \cdot \rrbracket_{\mathcal{A}, I} : \mathbf{E} \times \mathbf{A} \times \mathbf{I} \mapsto \mathbf{D}$$

takes an algebra and an interval to evaluate an expression and returns a domain element. Given an algebra \mathcal{A} and an interval I , an expression e evaluates to $\llbracket e \rrbracket_{\mathcal{A}, I}$. For formulas $\varphi \in \mathbf{F}$, we shall often write $\mathcal{A}, I \models \varphi$ (“ \mathcal{A} and I model φ ”), where

$$\mathcal{A}, I \models \varphi \quad \text{iff} \quad \llbracket \varphi \rrbracket_{\mathcal{A}, I} = \text{tt} .$$

Given an algebra \mathcal{A} , a formula φ is called valid (abbreviated by $\mathcal{A} \models \varphi$), if $\mathcal{A}, I \models \varphi$ for all I .

The semantics of the different expressions e , i.e. the value of the evaluation function $\llbracket e \rrbracket_{\mathcal{A}, I}$, is recursively defined in the following sections, starting with variables (Sect. 2.2.2), operators (Sect. 2.2.3) and quantifiers (Sect. 2.2.4), and including the ITL (Sect. 2.2.5) and LTL operators (Sect. 2.2.6). The semantics of sequential programs and procedures can be found in Sections 2.2.7 and 2.2.9. Separate Sections 2.3 - 2.8 follow which are dedicated to the semantics of the different operators for parallel programs. The whole chapter concludes with the important issue of substitution in Section 2.11.

If the algebra \mathcal{A} is not significant, it will be omitted in the following.

2.2.2 Variables

Definition 9 (*Semantics of variables*) The semantics of static variables X and dynamic or program variables $w \in \mathbf{Y} \cup \mathbf{Z}$ is defined as

$$\begin{aligned} \llbracket X \rrbracket_I &:= I(0)(X) \\ \llbracket w \rrbracket_I &:= I(0)(w) \\ \llbracket w' \rrbracket_I &:= I(0)'(w) \\ \llbracket w'' \rrbracket_I &:= I(0)''(w) \end{aligned}$$

In the last state, i.e. if I is empty, the value of a primed or double primed variable is no different from the unprimed variable (compare Def. 7). It is assumed that after a system has terminated, the variables do not change.

2.2.3 Operators

An algebra \mathcal{A} maps operator symbols to functions. To evaluate an application of operator f , the parameters are evaluated and the corresponding function $f_{\mathcal{A}}$ is applied.

Definition 10 (*Semantics of operator application*) The semantics of the application of an operator is defined as

$$\llbracket f(e_1, \dots, e_n) \rrbracket_{\mathcal{A}, I} := f_{\mathcal{A}}(\llbracket e_1 \rrbracket_{\mathcal{A}, I}, \dots, \llbracket e_n \rrbracket_{\mathcal{A}, I})$$

We assume that every algebra assigns the standard semantics to the boolean operators true, false, \neg , \wedge , \vee , \rightarrow , and \leftrightarrow and operators for natural numbers 0, succ, and $+$.

Definition 11 (*Semantics of equations*) The semantics of equations is defined as

$$I \models (e_1 = e_2) \quad \text{iff} \quad \llbracket e_1 \rrbracket_I = \llbracket e_2 \rrbracket_I$$

2.2.4 Quantifiers

In our logic it is possible to quantify both static and dynamic or program variables. Whereas for the semantics the difference between quantifying a static, dynamic or program variable is irrelevant, there is subtle difference in the proof method. In order to highlight their different usage in the proof method, separate quantors \exists and \exists (resp. \forall and \forall) are defined.

Definition 12 (*Semantics of quantifiers*) Let $v \in \mathbf{X} \cup \mathbf{Y}$, $w \in \mathbf{X} \cup \mathbf{Y} \cup \mathbf{Z}$. The semantics of the basic quantifiers is defined as

$$\begin{aligned} I \models \exists v. \varphi & \quad \text{iff} \quad \text{there exists } I_0 \text{ with } I_0 =_v I \text{ and } I_0 \models \varphi \\ I \models \exists x. \varphi & \quad \text{iff} \quad \text{there exists } I_0 \text{ with } I_0 =_x I \text{ and } I_0 \models \varphi \end{aligned}$$

where $I_1 =_w I_2$ (“ I_1 is equal to I_2 modulo w ”) is defined as

$$\begin{aligned} |I_1| &= |I_2| \\ \text{and } I_1(i)(w_0) &= I_2(i)(w_0) \text{ for all } w_0 \neq w, i \leq |I_1| \\ \text{and } I_1(i)'(w_0) &= I_2(i)'(w_0) \text{ for all } w_0 \neq w, i < |I_1| \end{aligned}$$

Further quantifiers are defined as abbreviations:

$$\begin{aligned} \forall v. \varphi & \quad := \neg \exists v. \neg \varphi \\ \forall x. \varphi & \quad := \neg \exists x. \neg \varphi \end{aligned}$$

2.2.5 ITL operators

The following ITL operators are taken from [22].

Definition 13 (*Semantics of ITL operators*) The semantics of the basic ITL operators is defined as

$$\begin{aligned} I \models \varphi; \psi & \quad \text{iff} \quad \text{there exists } n \leq |I| \text{ with } I|^n \models \varphi \text{ and } I|_n \models \psi \\ \text{or } |I| &= \infty \text{ and } I \models \varphi \end{aligned}$$

$$\begin{aligned}
I \models \varphi^* \quad \text{iff} \quad & |I| = 0 \\
& \text{or there exist } 0 = n_0 < n_1 < \dots < n_m < |I| \\
& \text{with } I|_{n_i}^{n_{i+1}} \models \varphi \text{ for all } 0 \leq i < m \\
& \text{and } I|_{n_m} \models \varphi \\
& \text{or } |I| = \infty \\
& \text{and there exist infinitely many } 0 = n_0 < n_1 < \dots \\
& \text{with } I|_{n_i}^{n_{i+1}} \models \varphi \text{ for all } 0 \leq i \\
I \models \mathbf{step} \quad \text{iff} \quad & |I| = 1
\end{aligned}$$

Further ITL operators are defined as abbreviations:

$$\begin{aligned}
\mathbf{more} & \quad \equiv \quad \mathbf{step}; \text{ true} \\
\mathbf{last} & \quad \equiv \quad \neg \mathbf{more} \\
\mathbf{inf} & \quad \equiv \quad \text{true}; \text{ false} \\
\mathbf{finite} & \quad \equiv \quad \neg \mathbf{inf} \\
\mathbf{finally } \varphi & \quad \equiv \quad \text{true}; (\mathbf{last} \wedge \varphi)
\end{aligned}$$

The chop operator $\varphi; \psi$ either requires φ to eventually terminate and ψ to be executed afterwards, or φ to run infinitely long. Note that in the first case the two intervals $I|_n$ and $I|_m$ overlap in state $I(n)$; the final state of φ is the initial state for the execution of ψ . This is exactly how sequential composition of programs should behave. The star operator is used to model loops, which either terminate directly or are repeated finitely or infinitely often. It is possible that in the finite case, the last cycle does not terminate. A cycle takes at least one step! Together with the **finally** operator, it is possible to derive while loops as follows:

$$\mathbf{while } \psi \mathbf{ do } \varphi \quad \equiv \quad (\psi \wedge \varphi)^* \wedge \mathbf{finally } \neg \psi$$

See below for further details on the semantics of sequential programs.

2.2.6 LTL operators

Most of the LTL operators can be derived in ITL with the exception of **until**, which is here defined as a basic operator.

Definition 14 (*Semantics of LTL operators*) *The semantics of the basic LTL operator is defined as*

$$\begin{aligned}
I \models \varphi \mathbf{until } \psi \quad \text{iff} \quad & \text{there exists } n \leq |I| \\
& \text{with } I|_n \models \psi \\
& \text{and } I|_m \models \varphi \text{ for all } 0 \leq m < n
\end{aligned}$$

Further LTL operators are defined as abbreviations:

$$\begin{aligned}
\diamond \varphi &::= \text{true } \mathbf{until} \varphi \\
\square \varphi &::= \neg \diamond \neg \varphi \\
\blacksquare \varphi &::= \square (\neg \mathbf{last} \rightarrow \varphi) \\
\varphi \mathbf{unless} \psi &::= \varphi \mathbf{until} \psi \vee \square \varphi \\
\circ \varphi &::= \mathbf{step}; \varphi \\
\bullet \varphi &::= \neg \circ \neg \varphi
\end{aligned}$$

Because intervals can also be finite, the next operator comes in two flavours. The strong next $\circ \varphi$ requires that there is a next step satisfying φ , the weak next $\bullet \varphi$ only states that if there is a next step, it must satisfy φ .

The always operator $\square \varphi$ evaluates φ on every postfix of an interval. Similarly, operators \boxplus for every prefix and \boxminus for every infix can be derived, however, these variants are of no relevance here. Instead, a “weak” always operator $\blacksquare \varphi$ which requires φ to always hold except for the last state is useful to express that every transition satisfies a certain condition. For example $\blacksquare n' < n$ states that every system transition decreases n – in the last state n' evaluates to the same value as n thus violating the desired property.

2.2.7 Sequential programs

[23] shows how to derive program constructs in ITL. However, assignments in ITL only restrict the assigned variables, whereas program assignments also leave other program variables unchanged. This is known as a frame assumption. In our logic, we define an explicit frame assumption which states that a system transition leaves all but a selection of program variables unchanged.

Definition 15 (*Semantics of frame assumptions*) *The semantics of frame assumptions $\{[x_1, \dots, x_n]\}$ is defined as*

$$I \models [x_1, \dots, x_n] \quad \text{iff} \quad \text{for all } x \in \mathbf{Z} \setminus \{x_1, \dots, x_n\}, \\
I(0)'(x) = I(0)(x)$$

A frame assumption can be viewed as an infinite conjunction

$$[x_1, \dots, x_n] \equiv \bigwedge_{x \notin \{x_1, \dots, x_n\}} x' = x .$$

As it turns out, the frame assumption raises a number of problems starting with the definition of free variables and substitution in Section 2.11. On the other hand it is very useful to naturally define system behaviour. Furthermore, it is essential for our

abstraction technique. It is important to note that with the definition above it is only possible to express frame assumptions for system transitions; the environment transition – the relation between primed and double primed variables – cannot be restricted to such an extent.

In the style of ITL, the semantics of all sequential program constructs can be derived as follows. As sequential programs, we have taken all the program constructs from Dynamic Logic [13], where in turn the language is similar to Pascal.

Definition 16 (*Semantics of sequential programs*) *Constructs for sequential programs are derived as follows:*

$$\begin{aligned}
x := e &::= x' = e \wedge [x] \wedge \circ \text{last} \\
x := ? &::= [x] \wedge \circ \text{last} \\
\text{skip} &::= [] \wedge \circ \text{last} \\
\text{var } x = e \text{ in } \varphi &::= \square x' = x \\
&\quad \wedge \exists X. X = e \wedge \exists x. x = X \wedge \varphi \wedge \square x'' = x' \\
&\quad X \notin \text{free}(e, \varphi) \\
\text{var } x = ? \text{ in } \varphi &::= \square x' = x \wedge \exists x. \varphi \wedge \square x'' = x' \\
\text{if } \psi \text{ then } \varphi_1 \text{ else } \varphi_2 &::= (\psi \rightarrow \varphi_1) \wedge (\neg \psi \rightarrow \varphi_2) \\
\text{if } \psi \text{ then } \varphi &::= \text{if } \psi \text{ then } \varphi \text{ else last} \\
\text{while } \psi \text{ do } \varphi &::= (\psi \wedge \varphi)^* \wedge \text{finally } \neg \psi \\
\text{abort} &::= \text{while true do skip}
\end{aligned}$$

Executing an assignment requires exactly one step. The evaluation of conditions and the initialisation of local variables execute in no time. In this aspect, our language is artificial, leaving us with full control of the number of steps it takes to execute a program.

The semantics of local variables includes a number of interesting details: the local variable is quantified ($\exists x$); in addition, the environment cannot access the local variable ($\square x'' = x'$). Also, the global value of the variable is unchanged ($\square x' = x$). Initialisation is nontrivial in the case of x occurring in expression e . Occurrences of x in e refer to its global value. Therefore, the expression is evaluated first, and its value is stored in a fresh static variable X . The local variable x is then initialised with X ($x = X$). An alternative would be to rename the local variable x as a fresh program variable which does not occur in e . However, a fresh program variable cannot be obtained in the general case as is discussed in Section 2.11.

The **abort** from Dynamic Logic is interpreted as a nonterminating loop doing nothing.

2.2.8 Parallel programs

The semantics of the different parallel operators is an important part of this work. Therefore, separate sections have been dedicated to them. Synchronisation with **await** is defined in Sect. 2.3, the semantics of the interleaving operator is discussed in Sect. 2.4 and is finally defined in Section 2.6. Dijkstra's nondeterministic choice can be found in Sect. 2.7 followed by the semantics of atomic steps (Sect. 2.8).

2.2.9 Procedures

Procedures are used to modularise systems. An algebra maps procedure symbols to relations. Different from functions, a procedure takes a sequence of values for its reference parameters, the reference parameters being variables which are shared with the environment. The special variable `blk` is an implicit reference parameter of every procedure. The global values of all but the reference parameters remain unchanged; thus, procedures integrate a frame assumption. Execution of a procedure can be nondeterministic. Also, the procedure may sometimes terminate after a number of steps or run forever.

Definition 17 (*Semantics of procedures*)

$$\begin{aligned} \mathcal{A}, I \models \text{proc}(e_1, \dots, e_n; x_1, \dots, x_m) \\ \text{iff } \text{proc}_{\mathcal{A}} \left(\llbracket e_1 \rrbracket_{\mathcal{A}, I}, \dots, \llbracket e_n \rrbracket_{\mathcal{A}, I}, \left(d_{x_1}^{(i)} \times \dots \times d_{x_m}^{(i)} \times d_{\text{blk}}^{(i)} \right)_{i=0}^{2 \cdot |I|} \right) \\ \text{and } \mathcal{A}, I \models \square [x_1, \dots, x_m, \text{blk}] \end{aligned}$$

where

$$d_x^{(i)} := \begin{cases} I(i/2)(x) & \text{if } i \text{ is even} \\ I((i-1)/2)'(x) & \text{otherwise} \end{cases}$$

The length of the stream of values for the reference parameters is $2 \cdot |I| + 1$. This is because unprimed and primed values occur separately within the stream: the relation specifies, how the procedure reacts to input from the environment in each step!

Example Consider an example procedure $P(n; m)$ with value parameter n and a reference parameter m . We "implement" the procedure as follows:

$$P(n; m) \leftrightarrow \text{var } n = n \text{ in } m := 0; m := n$$

Under this equivalence, the formula $P(n+1; m)$ is satisfied by the following intervals (with $n_i, m_i \in \mathbf{D}_{\text{nat}}, b_i \in \mathbf{D}_{\text{bool}}$):

	σ_0	σ'_0	σ_1	σ'_1	σ_2
<code>n</code>	n_0	n_0	n_0	n_0	n_0
<code>m</code>	m_0	0	m_1	$n_0 + 1$	m_2
<code>blk</code>	b_0	b_0	b_1	b_1	b_2

As can be seen, the sequence of values for the variable m arbitrarily changes from a primed state to the following unprimed state. The environment is not restricted and thus behaves nondeterministically. After m has been assigned 0 in σ'_0 , the environment again changes the variable to an unknown value m_1 . This value is the input to the procedure for the next step. The value parameter n is stored in a local variable n which cannot be accessed by the environment. Thus, the second assignment refers to the original value of n .

2.2.10 Systems

Definition 18 (*Semantics of system operators*) *The semantics of the diamond operator is defined as*

$$I \models \langle \varphi \rangle \psi \quad \text{iff} \quad \begin{array}{l} \text{there exists } I_0 \\ \text{with } I_0(0) = I(0) \\ \text{and } I_0 \models \varphi \text{ and } I_0 \models \psi \end{array}$$

The semantics of the box operator can be derived as follows:

$$[\varphi] \psi \quad :\equiv \quad \neg \langle \varphi \rangle \neg \psi$$

The diamond operator claims that there is a possibility to continue from the initial state $I(0)$ which both satisfies φ and ψ . The diamond and box operators can be used to quantify intervals; while $\langle \varphi \rangle \psi$ states that there is an interval satisfying both formulas, $[\varphi] \psi$ requires all intervals satisfying φ to also satisfy ψ .

Example Formula

$$\langle n := 0 \vee n := 1 \rangle \diamond n' = 1$$

reads: “there is an execution of the nondeterministic program $n := 0 \vee n := 1$ where eventually $n' = 1$ ”. The interval $I = (n_0 = 1, n'_0 = 1, n_1 = 1)$ satisfies both $n := 1$ and $\diamond n' = 1$, therefore the diamond formula is valid.

However, there is an important difference between our system operators and the path operators of Concurrent Tree Logic (CTL). While formulas in our logic are valid, if they are satisfied by all intervals $I \in \mathbf{I}$, formulas in CTL are valid only, if they are satisfied by all intervals $I \in M$ for all subsets $M \subseteq \mathbf{I}$. The following example illustrates the difference.

Example Consider

$$(\mathbf{A} \diamond n = 0) \rightarrow (\mathbf{A} \square n = 0)$$

which reads: “if for all paths $I \in M$ (with system $M \subseteq \mathbf{I}$) eventually a state satisfies $n = 0$, then on all paths $\square n = 0$ holds”. This formula is not valid, because for $M = \{(n_0 = 0, n'_0 = 1, n_1 = 1)\}$ the precondition is satisfied, but the postcondition does not hold. In contrast,

$$([\text{true}] \diamond n = 0) \rightarrow ([\text{true}] \square n = 0)$$

is a valid property, because the precondition “all paths $I \in \mathbf{I}$ satisfy $\diamond n = 0$ ” gives a contradiction: the interval $I = (n_0 = 1, n'_0 = 1, n_1 = 1)$ does not satisfy $\diamond n = 0$.

The box operator quantifies over all possible continuations starting from a fixed initial state. In order to quantify over all intervals independent from the initial state, we shall use a combination of box and weak next operators.

Corollary 1 (*System operator with next*) *Formula $[\text{true}] \bullet \varphi$ with a combination of box and weak next operator satisfies the following property:*

$$I \models [\text{true}] \bullet \varphi \quad \Leftrightarrow \quad \text{for all } I_0, I_0 \models \varphi$$

Directly follows from the Semantics of $[\]$ and \bullet . □

2.3 Synchronisation

Parallel programs communicate using shared variables. In order to synchronise execution, the operator **await** φ can be used. As long as condition φ is not satisfied, the process guarded with the await statement is blocked and does nothing. Blocking is modelled using a special variable `blk`. The semantics of the operator can be given as follows.

Definition 19 (*Semantics of synchronisation*) *The semantics of synchronisation with **await** φ is defined as*

$$\begin{aligned} \mathbf{blocked} &::\equiv \text{blk}' \neq \text{blk} \\ \mathbf{await} \varphi &::\equiv \mathbf{while} \neg \varphi \mathbf{do} (\lceil \text{blk} \rceil \wedge \mathbf{blocked} \wedge \circ \mathbf{last}) \end{aligned}$$

The **await** operator behaves like a loop waiting for the condition to be satisfied. While the operator waits, no variable is changed except variable `blk` which is toggled. In other words, a process guarded with an **await** operator actively waits for the environment to satisfy its condition. We consider toggling of the truth value of `blk` ($\text{blk}' \neq \text{blk}$) as an indicator for a program to be blocked instead of referring to the concrete value of `blk` ($\text{blk}' = \text{true}$). In this way, it is not necessary to initialise the variable. Furthermore, an assignment $x = e$, which leaves all variables except `x` unchanged, is automatically not blocked.

An interval is considered to be blocked, if and only if the interval is nonempty, and variable `blk` toggles in the first system transition. The following definition formally captures this notion.

Definition 20 (*Blocked interval*) An interval I terminates, iff $|I| = 0$.

An interval I is blocked, iff

- I does not terminate, and
- $I(0)'(\text{blk}) \neq I(0)(\text{blk})$.

An interval I is active, iff I is not blocked. It progresses, iff it is active and does not terminate.

2.4 Interleaving

Intuitively, interleaving two processes is to execute either a transition of the first or the second process. To interleave not only programs, but also formulas, the semantics of interleaving is reduced to the interleaving of intervals $I_1 \parallel I_2$. The definition of interleaving formulas can then be defined

$$I \models \varphi \parallel \psi \quad \text{iff} \quad \begin{array}{l} \text{there exist } I_1, I_2 \\ \text{with } I \in \llbracket I_1 \parallel I_2 \rrbracket \text{ and } I_1 \models \varphi \text{ and } I_2 \models \psi . \end{array}$$

An interval I satisfies an interleaving $\varphi \parallel \psi$, if and only if I is a possible result of the interleaving of two intervals I_1 and I_2 satisfying φ and ψ . The set of intervals $\llbracket I_1 \parallel I_2 \rrbracket$ remains to be defined.

In an interval, transitions are modelled as relations between states, the relation from σ_i to σ'_i being a system transition and the relation from σ'_i to σ_{i+1} being an environment transition. To better illustrate the idea of interleaving intervals, blocking is not considered in the following section, i.e. each system transition is assumed not to be blocked.

2.4.1 Interleaving without blocking

The basic idea of how to interleave two intervals I_1 and I_2 , where both intervals are never blocked, is illustrated in Figure 2.2. Interleaving intervals is almost straightforward: a possible interval from the set of intervals $I_1 \parallel I_2$ alternately contains system transitions from I_1 and I_2 . However, two special cases require further attention.

- (*Environment transitions*) System transitions are interleaved by including states σ_i and σ'_i from either interval I_1 or I_2 in the resulting interval. An environment transition is the relation from σ'_i to σ_{i+1} . In the resulting interval, the environment transition of one process is established between σ'_i and σ_j with j not necessarily being equal to $i + 1$. Step i is the state where the system transition preceding the

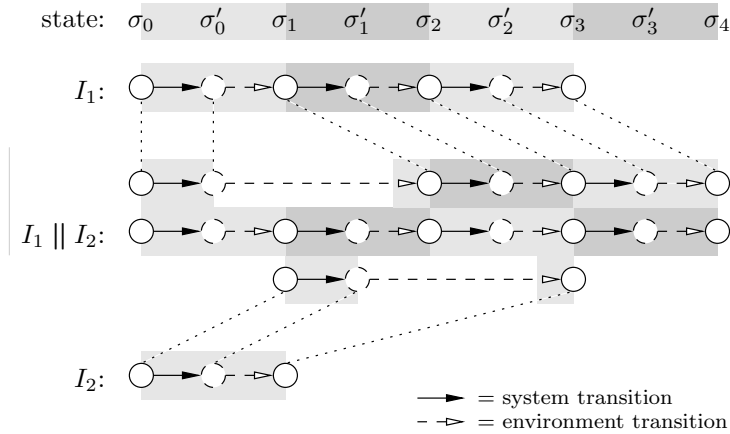


Figure 2.2: Interleaving intervals without blocking

environment transition has been executed, and j is the state where the same process is again considered. As a consequence, the environment transitions of one process enclose all of the transitions of the other process which have been interleaved. This ensures that – from the perspective of a single process, the processes running in parallel are considered part of its environment.

In Fig. 2.2, the first environment transition of I_1 is established between σ'_0 and σ_2 of the resulting interval. The inserted system transition of I_2 – together with transitions of the environment of the whole interleaving $I_1 \parallel I_2$ – must satisfy the environment transition of I_1 .

- (*Termination*) If the last state of an interval is reached (e.g., in Fig. 2.2 the terminating state of I_2 in state σ_3 of $I_1 \parallel I_2$), execution immediately continues with a transition of the other interval (e.g., with the third system transition of I_1). This is as expected. However it is interesting to note that the last state need not immediately follow the last system transition of an interval. After the last system transition has been executed, it may still take a number of steps until the final state is considered. This is because the last system transition is always followed by a final environment transition which encloses zero or more system transitions of the interleaved interval.

2.4.2 Blocking

Interleaving is a bit more complicated, if explicit blocking is considered. Figure 2.3 contains an interval I_1 where the first two system transitions are blocked, i.e. the value of blk' differs from blk . If I_1 is interleaved with I_2 , the interval labelled with $I_1 \parallel I_2$ could result. The first transitions of the resulting interval are explained next.

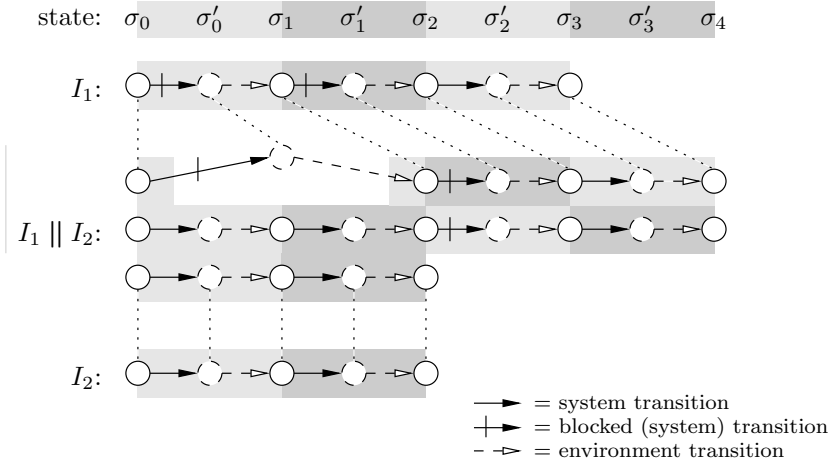


Figure 2.3: Interleaving intervals with blocking

- In the depicted case, a transition of I_1 is considered first. However, this transition is blocked and therefore I_2 is executed as well. It is important to note that both transitions are consumed; I_1 is *actively* waiting for its condition to be satisfied. The primed variables in state σ'_0 , however, are only determined by the transition of I_2 , state σ'_0 of I_1 is discarded. Though both transitions are consumed, it is the transition of I_2 which determines the settings of the variables in the next state.
- In the second step, a transition of I_2 is executed. This transition is not blocked, therefore I_1 is not considered.
- In the next step, interval I_1 again is considered and is still blocked. Therefore, I_2 is executed as well. I_2 terminates, which leaves the blocked transition of I_1 as only possible continuation. As a consequence the transition in the resulting interval is also blocked.

2.4.3 Recursive equations for interleaving intervals

To arrive with a formal semantics, it must be noted first that interleaving two intervals results in a set of intervals

$$. \parallel . : \mathbf{I} \times \mathbf{I} \rightarrow \mathcal{P}^{\mathbf{I}},$$

the interval of Fig. 2.3 only being an example of many possibilities. The scheduler either tries to execute a transition of the first or the second interval in each step. To separate these two cases, a function $\parallel^<$ with

$$. \parallel^< . : \mathbf{I} \times \mathbf{I} \rightarrow \mathcal{P}^{\mathbf{I}}$$

is introduced. $I_1 \parallel^< I_2$ executes the interval I_1 first. With this function, the resulting set of interleaving two intervals is the union of two cases

$$I_1 \parallel I_2 := (I_1 \parallel^< I_2) \cup (I_2 \parallel^< I_1) .$$

For $I_1 \parallel^< I_2$ again three cases can be distinguished. Either, the considered interval I_1 terminates, or its first transition is either blocked or not blocked. These cases are examined next.

1. (*Termination*) If $|I_1| = 0$, i.e. the considered interval terminates, then

$$I_1 \parallel^< I_2 = \begin{cases} \{I_2\}, & \text{if } I_1(0) = I_2(0) \\ \emptyset, & \text{otherwise .} \end{cases}$$

If I_1 terminates, interval I_2 is the only possible continuation. However, the final state of I_1 must be equal to the initial state of I_2 ; state $I_1(0)$ is the one with which execution of I_2 continues. Otherwise, these two intervals cannot be interleaved and an empty set of intervals results. Remember that the intervals represent a single possibility to execute the interleaved programs and I_2 assumes one of many concrete initial states.

2. (*Progress*) If $|I_1| > 0$ and I_1 is not blocked, then

$$I_1 \parallel^< I_2 = (I_1(0), I_1(0)') \oplus (I_1|_1 \parallel I_2) .$$

Function \oplus prefixes the first transition of I_1 to each interval in the set of intervals $I_1|_1 \parallel I_2$ resulting from interleaving the rest of I_1 with I_2 . The function \oplus is defined

$$(\sigma, \sigma') \oplus \mathcal{I} := \{(\sigma, \sigma', \sigma_0, \dots) \mid (\sigma_0, \dots) \in \mathcal{I}\} .$$

3. (*Blocking*) Finally, if I_1 is blocked, then

$$I_1 \parallel^< I_2 = \begin{cases} (I_2(0), I_2(0)') \oplus (I_1|_1 \parallel I_2|_1), & \text{if } |I_2| > 0 \text{ and } I_1(0) = I_2(0) \\ \emptyset, & \text{otherwise .} \end{cases}$$

Interval I_1 is blocked. Therefore a transition of I_2 – if any – is executed instead. If the initial states of both intervals are not compatible, i.e. $I_1(0) \neq I_2(0)$, the result is empty. Note, that in the case of I_1 being blocked, I_2 is assumed not to terminate. This is an optimisation: the case, where I_2 terminates while I_1 is blocked is already contained in the other set of intervals $I_2 \parallel^< I_1$.

These recursive equations do not properly define the semantics of interleaving intervals as for infinite intervals the recursion does not terminate. Furthermore, the scheduler for interleaving should be fair, ensuring that each process is always eventually considered. A declarative semantics describing how to fairly interleave infinite intervals is rather awkward and difficult to understand. It is more intuitive to denote the semantics in an operational style. Therefore, in the next section a notation called Structural Operational Semantics (SOS) is introduced before revisiting the semantics of interleaving intervals in Section 2.6.

2.5 SOS notation

Structural Operational Semantics (SOS) is a technique to generate a labelled transition system from a set of transition rules, a so-called transition system specification (TSS). We use SOS rules to intuitively describe the semantics of some of the operators where an operational semantics is more intuitive. SOS originates from [26]. Here, we stick to [2]. Our transition rules adhere to the following format:

$$\frac{t_1 \xrightarrow{a} t'_1 \quad \dots \quad t_k \xrightarrow{a} t'_k \quad t_{k+1} P_{k+1} \quad \dots \quad t_n P_n}{t \xrightarrow{a} t'}$$

The premises are either transitions $t_i \xrightarrow{a} t'_i$ or predicates $t_i P_i$. The predicates are used to formulate side conditions; they never occur as conclusions of transition rules. Furthermore, we do not require negative premises, which would otherwise complicate the interpretation of TSSs as described in [2].

Example The relation $I_1 \parallel I_2 \xrightarrow{\sigma, \sigma'} t'$ defines the operational semantics of interleaving two intervals by describing how to execute a single first step: if the relation holds, then the interleaving $I_1 \parallel I_2$ can progress from state σ to state σ' with the system in the next state being t' .

The relation is true, if and only if it can be derived from the transition rules in the transition system specification. An example rule is as follows:

$$\frac{I_1 \xrightarrow{\sigma_1, \sigma'_1} I'_1 \quad I_1 \text{ progresses}}{I_1 \parallel I_2 \xrightarrow{\sigma_1, \sigma'_1} I'_1 \parallel I_2} \text{ } \textit{ilv prgr}$$

In this context, variables t represent terms of a given language. We intend to define an operational semantics for the interleaving of intervals as well as the nondeterministic choice between intervals. Therefore, the BNF grammar for the language we use is (with $I, I_1, I_2 \in \mathbf{I}$, $n \in \mathbb{N}$):

$$t ::= \emptyset \mid I \mid (\varepsilon_1 \parallel_n I_1 \mid \varepsilon_2 \parallel_n I_2) \mid (I_1 \parallel I_2)$$

The basic elements or constants of our language are intervals, explicitly describing runs of an abstracted system. The grammar defines a very simple language and contains no recursion. An auxiliary operator $I_1 \parallel_n I_2$ is used to define fair interleaving, and an additional symbol \emptyset represents the terminated system. $I_1 \parallel I_2$ is used in Sect. 2.7 to define the choice operator.

The set of actions consists of pairs of states $\Sigma \times \Sigma$. The notion of traces from [2] is adapted to our needs. Especially, we take intervals as sequences of actions and also consider infinite traces.

Definition 21 (*SOS traces*) Let t be an SOS term. Let $(t_i)_{i=0}^{\bar{n}}$ be a finite or infinite sequence of SOS terms.

$$\begin{aligned} I \in \llbracket (t_i)_{i=0}^{\bar{n}} \rrbracket & \text{ iff } & \bar{n} = |I| + 1 \text{ and } (t_{\bar{n}+1} = \emptyset \text{ or } \bar{n} = \infty) \\ & & \text{and } t_i \xrightarrow{I(i), I(i)'} t_{i+1} \text{ for all } i < \bar{n} \\ I \in \llbracket t \rrbracket & \text{ iff } & \text{there exists } (t_i)_{i=0}^{\bar{n}} \text{ with } t = t_0 \text{ and } I \in \llbracket (t_i) \rrbracket \end{aligned}$$

An interval is a trace of t , if and only if there is a sequence of SOS terms (t_i) starting with t , where the neighbouring terms t_i and t_{i+1} adhere to the SOS relation. For finite traces, the last SOS term must equal the terminated system \emptyset .

The following definition contains a set of transition rules describing how to execute single intervals. Additional rules in sections 2.6 and 2.7 define the operational semantics of interleaving and nondeterministic choice.

Definition 22 (*Operational semantics of executing intervals*) The operational semantics of execution an interval I is defined according to the following SOS rules.

$$\frac{|I| = 0}{I \xrightarrow{I(0), I(0)'} \emptyset} \text{ trm} \quad \frac{|I| > 0}{I \xrightarrow{I(0), I'(0)'} I|_1} \text{ stp}$$

For further definitions and proofs it is convenient to define a special postfix operator for intervals which returns the SOS term \emptyset , if the length of the interval is exceeded.

Definition 23 (*Postfix of SOS term*) The postfix of an interval (in the role of an SOS term) is defined as follows:

$$I|_i^\emptyset := \begin{cases} I|_i & \text{if } i \leq |I| \\ \emptyset & \text{otherwise} \end{cases}$$

With this postfix operator, an SOS step of executing a single interval is equivalent to a formula with a single case.

Corollary 2 (*SOS relation for intervals*)

$$I \xrightarrow{\sigma, \sigma'} t_{i+1} \Leftrightarrow \sigma = I(0) \text{ and } \sigma = I(0)' \text{ and } t_{i+1} = I|_1^\emptyset$$

See Appendix C.1.1.

□

2.6 Interleaving intervals

Using SOS rules, we first define a (more intuitive) operational semantics of interleaving arbitrary formulas. Afterwards, we derive a declarative semantics which is, though more difficult to understand, sometimes better for reasoning about interleaving. As basic interleaving operator, we use $l_1 :: \varphi \parallel^< l_2 :: \psi$ which is the labelled left interleaving of φ and ψ . Labels l_1 and l_2 are required for induction (see Chapter 6).

2.6.1 Operational semantics

Definition 24 (*Operational semantics of interleaving*) *The semantics of left interleaving two intervals $l_1 :: \varphi \parallel^< l_2 :: \psi$ is defined as*

$$I \models l_1 :: \varphi \parallel^< l_2 :: \psi \quad \text{iff} \quad \begin{array}{l} \text{there exist } I_1, I_2, n \in \mathbb{N}_0 \\ \text{with } I \in \llbracket l_1 :: I_1 \parallel_{n+1}^< l_2 :: I_2 \rrbracket \\ \text{and } I_1 \models \varphi \text{ and } I_2 \models \psi \end{array}$$

where

$$\begin{array}{c} \frac{I_1 \xrightarrow{\sigma, \sigma'_1} \emptyset \quad I_2 \xrightarrow{\sigma, \sigma'_2} t'}{(l_1 :: I_1 \parallel_{n+1}^< l_2 :: I_2) \xrightarrow{\sigma, \sigma'_2} t'} \quad \text{ilvl lst} \\ \\ \frac{I_1 \xrightarrow{\sigma_1, \sigma'_1} I'_1 \quad I_1 \text{ progresses} \quad (\sigma_1) \not\models l_2}{(l_1 :: I_1 \parallel_{n+1}^< l_2 :: I_2) \xrightarrow{\sigma_1, \sigma'_1} (l_1 :: I'_1 \parallel_n^< l_2 :: I_2)} \quad \text{ilvl stp} \\ \\ \frac{I_1 \xrightarrow{\sigma, \sigma'_1} I'_1 \quad I_1 \text{ is blocked} \quad I_2 \xrightarrow{\sigma, \sigma'_2} I'_2}{(l_1 :: I_1 \parallel_{n+1}^< l_2 :: I_2) \xrightarrow{\sigma, \sigma'_2} (l_1 :: I'_1 \parallel_n^< l_2 :: I'_2)} \quad \text{ilvl blk} \\ \\ \frac{(l_2 :: I_2 \parallel_{n+1}^< l_1 :: I_1) \xrightarrow{\sigma, \sigma'} t'}{(l_1 :: I_1 \parallel_0^< l_2 :: I_2) \xrightarrow{\sigma, \sigma'} t'} \quad \text{ilvl swtch}_n, \quad \text{for } n \in \mathbb{N}_0 \end{array}$$

Other interleaving operators can be derived:

$$\begin{array}{l} l_1 :: \varphi \parallel^> l_2 :: \psi \quad \equiv \quad l_2 :: \psi \parallel^< l_1 :: \varphi \\ l_1 :: \varphi \parallel l_2 :: \psi \quad \equiv \quad l_1 :: \varphi \parallel^< l_2 :: \psi \vee l_1 :: \varphi \parallel^> l_2 :: \psi \\ l_1 :: \varphi \parallel_b^< l_2 :: \psi \quad \equiv \quad l_1 :: (\mathbf{blocked} \wedge \circ \varphi) \parallel^< l_2 :: \psi \\ l_1 :: \varphi \parallel_b^> l_2 :: \psi \quad \equiv \quad l_1 :: \varphi \parallel^> l_2 :: (\mathbf{blocked} \wedge \circ \psi) \end{array}$$

The considered interval either terminates (*ilvl trm*), progresses (*ilvl stp*) or is blocked (*ilvl blk*). Finally, after executing $n + 1$ steps of one interval I_1 , the scheduler switches to the other interval I_2 , executing a nondeterministic but positive and finite number of steps of I_2 (*ilvl swtch_n*). The marker l_2 must be false in a state where the interval I_2 is

deferred while the other interval I_1 progresses. The default marker false trivially fulfils this requirement.

2.6.2 Declarative semantics

The declarative semantics for interleaving is based on the notion of a schedule. A schedule p is defined as a special marker to represent the decision of the scheduler which one of the two processes to execute next.

Definition 25 (*Scheduling sequences*) A schedule $p \in \{1, 2\}$ marks which process (process 1 or 2) is scheduled for execution.

A (finite or infinite) sequence of schedules $(p_i)_{i=0}^{\bar{n}}$ is called a scheduling sequence.

A scheduling sequence $(p_i)_{i=0}^{\bar{n}}$ is fair, if and only if the sequence is finite, or for all i , there exists $j > i$ with $p_j \neq p_i$.

A scheduling sequence formalises the behaviour of the scheduler in every state. The operational semantics of interleaving is fair and therefore a scheduling sequence where every process is always eventually considered is said to be fair.

With a scheduling sequence, two intervals I_1 and I_2 can be interleaved deterministically. The interleaving results in sequences of intervals $(I_1^{(i)})$ and $(I_2^{(i)})$ which represent the progress of the two processes in every state. Because processes can be terminated or blocked it is possible that a process different from the scheduled process p_i is executed in state i . We therefore derive an additional set of actual schedules π_i from a schedule p_i . The set of actual schedules contains the markers of the processes which are actually executed in state i .

Definition 26 (*Scheduling intervals*) Scheduling two intervals I_1 and I_2 with a given scheduling sequence $(p_i)_{i=0}^{\bar{n}}$ results in a sequence of sets of actual schedules $(\pi_i)_{i=0}^{\bar{n}+1}$ and two sequences of intervals $(I_1^{(i)})_{i=0}^{\bar{n}+1}$ and $(I_2^{(i)})_{i=0}^{\bar{n}+1}$ which are recursively defined as follows.

$$\begin{aligned}
 I_1^{(0)} &= I_1 \\
 I_1^{(i+1)} &= \begin{cases} I_1^{(i)}|_1^\emptyset, & \text{if } 1 \in \pi_i \\ I_1^{(i)}, & \text{otherwise} \end{cases} \\
 I_2^{(0)} &= I_2 \\
 I_2^{(i+1)} &= \begin{cases} I_2^{(i)}|_1^\emptyset, & \text{if } 2 \in \pi_i \\ I_2^{(i)}, & \text{otherwise} \end{cases}
 \end{aligned}$$

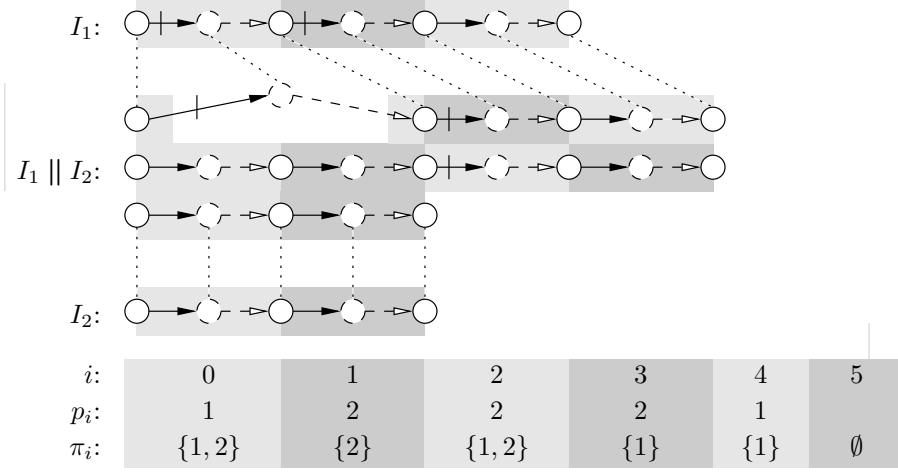


Figure 2.4: Schedules for interleaving intervals

$$\pi_i = \begin{cases} \emptyset & \text{if } I_1^{(i)} = \emptyset \text{ and } I_2^{(i)} = \emptyset \\ \{2\}, & \text{if } I_1^{(i)} = \emptyset \\ \{1\}, & \text{if } I_2^{(i)} = \emptyset \\ \{1\}, & \text{if } p_i = 1 \text{ and } I_1^{(i)} \text{ progresses} \\ \{2\}, & \text{if } p_i = 2 \text{ and } I_2^{(i)} \text{ progresses} \\ \{1, 2\}, & \text{otherwise} \end{cases}$$

Fig. 2.4 illustrates the sequence of sets of actual schedules $(\pi_i)_{i=0}^{\bar{n}+1}$. The figure contains a situation of interleaving two intervals which has already been discussed in Figure 2.3. Assume that initially $p_0 = 1$, i.e. process 1 is scheduled for execution. However, its first transition is blocked. Therefore, a transition of the second process is also executed and $\pi_0 = \{1, 2\}$. In the next state, the schedule $p_1 = 2$. The second process progresses and therefore $\pi_1 = \{2\}$. The second process is also scheduled for execution in the next state ($p_2 = 2$). It terminates and thus, also a transition of the first is immediately considered ($\pi_2 = \{1, 2\}$). In the following state, $p_3 = 2$ still holds. However, process 2 has terminated, and therefore the first process is actually executed, i.e. $\pi_3 = \{1\}$. After both processes have terminated in state 5, the set of actual schedules is empty ($\pi_5 = \emptyset$).

Using the derived sequences of the definition above, a declarative semantics of interleaving can be defined.

Lemma 1 (Declarative semantics of labelled left interleaving) *Given an interval I with $\bar{n} = |I|$. $I \models l_1 :: \varphi \parallel^< l_2 :: \psi$ if and only if there exist intervals I_1, I_2 with $I_1 \models \varphi$ and*

$I_2 \models \psi$ and a fair scheduling sequence $(p_i)_{i=0}^{\bar{n}}$ with $p_0 = 1$, $\bar{n} = \infty$ or $\pi_{\bar{n}+1} = \emptyset$, and for all $i \leq \bar{n}$ all of the following holds.

1. $\pi_i \neq \emptyset$,
2. if $1 \in \pi_i$ then $I(i) = I_1^{(i)}(0)$,
3. if $2 \in \pi_i$ then $I(i) = I_2^{(i)}(0)$,
4. $I(i)' = \begin{cases} I_1^{(i)}(0)' , & \text{if } \pi_i = \{1\} \\ I_1^{(i)}(0)' , & \text{if } \pi_i = \{1, 2\} \text{ and } I_1^{(i)} \text{ progresses} \\ I_1^{(i)}(0)' , & \text{if } \pi_i = \{1, 2\} \text{ and } I_2^{(i)} \text{ terminates} \\ I_2^{(i)}(0)' , & \text{otherwise} \end{cases}$
5. if $I_1^{(i)} \neq \emptyset$ and $1 \notin \pi_i$ then $(I(i)) \models \neg l_1$, and
6. if $I_2^{(i)} \neq \emptyset$ and $2 \notin \pi_i$ then $(I(i)) \models \neg l_2$.

For left interleaving, process 1 is always scheduled in the first state ($p_0 = 1$). Furthermore, if the resulting interval is finite, then both processes must have terminated in state $\bar{n} + 1$. In the intermediate states,

- one of the processes must be executed ($\pi_i \neq \emptyset$),
- the unprimed state of the actually scheduled processes must match the unprimed state of the resulting interval,
- one of the primed states of the actually scheduled processes is chosen to be equal to the primed state of the resulting interval, and
- the resulting unprimed state must falsify the marker of a process, if the process has not terminated and is not actually scheduled for execution.

(Proof of Lemma 1) See Appendix C.1.2. □

While the operational semantics of Def. 24 is more intuitive, the declarative semantics is more useful to prove properties of interleaving. An example is the coincidence lemma of Section 2.11.

2.7 Dijkstra's choice operator

The basic idea of the choice operator $\varphi \sqcap \psi$ is to select the option φ or ψ which becomes active first. If both options are active at the same time, the choice is nondeterministic. The operator is taken from [19].

Example The two options of the choice operator

$$\begin{aligned} & (\text{await } 1 \leq n \wedge n \leq 3; \\ & \quad \dots) \\ \parallel & (\text{await } 2 \leq n; \\ & \quad \dots) \end{aligned}$$

are guarded with await conditions. As a consequence, they are blocked as long as the condition is not satisfied. The first option is chosen as soon as variable n is between 1 and 3. The second option is chosen as soon as variable n is greater or equal to two. While n is zero, no option is selectable, and the whole choice operator is blocked. If n first becomes 1, the first choice is taken. If n first becomes 2 or 3, then both options are selectable and the choice is nondeterministic.

Again, we use SOS rules to give an operational semantics of the choice operator.

Definition 27 (*Semantics of Dijkstra's choice operator*) The semantics of Dijkstra's choice operator $\varphi \parallel \psi$ is defined as

$$I \models \varphi \parallel \psi \quad \text{iff} \quad \begin{aligned} & \text{there exist } I_1, I_2 \\ & \text{with } I \in \llbracket I_1 \parallel I_2 \rrbracket \text{ and } I_1 \models \varphi \text{ and } I_2 \models \psi \end{aligned}$$

where

$$\begin{aligned} & \frac{I_1 \xrightarrow{\sigma, \sigma'} t' \quad I_1 \text{ is active}}{I_1 \parallel I_2 \xrightarrow{\sigma, \sigma'} t'} \text{ chs } 1 \quad \frac{I_2 \xrightarrow{\sigma, \sigma'} t' \quad I_2 \text{ is active}}{I_1 \parallel I_2 \xrightarrow{\sigma, \sigma'} t'} \text{ chs } 2 \\ & \frac{I_1 \xrightarrow{\sigma, \sigma'} I'_1 \quad I_2 \xrightarrow{\sigma, \sigma'} I'_2 \quad I_1 \text{ is blocked} \quad I_2 \text{ is blocked}}{I_1 \parallel I_2 \xrightarrow{\sigma, \sigma'} I'_1 \parallel I'_2} \text{ chs blk} \end{aligned}$$

The blocked choice operator can be derived:

$$\varphi \parallel^b \psi \quad \equiv \quad (\text{blocked} \wedge \varphi) \parallel (\text{blocked} \wedge \psi)$$

Three rules are used to choose between two given intervals I_1 and I_2 . If I_1 is initially active, i.e. it terminates or progresses, then the first choice can be executed (rule *chs 1*). If I_2 is initially not blocked, then also the second choice can be executed (*chs 2*). If both intervals are blocked, then the blocked transition of both intervals is consumed and the choice is deferred to the next state (*chs blk*). The derived operator $\varphi \parallel^b \psi$ is a shortcut for the case where both sub programs are blocked in the first state.

2.8 Atomic steps

Operator $\{\varphi\}$ can be used to execute program φ in a single step. Thus, more complex calculations are possible within a transition. The following semantics definition is such

that $\{\varphi\}$ takes exactly one step, the first and the last state of executing φ being the pre and post state of the resulting transition. If φ does not terminate, then no transition results. If φ is nondeterministic, then each (terminating) run of φ gives a transition. No environment interaction is possible in φ .

Definition 28 (*Semantics of atomic steps*) The semantics of an atomic step $\{\varphi\}$ is defined as

$$I \models \{\varphi\} \quad \text{iff} \quad \begin{array}{l} |I| = 1 \\ \text{and there exists } I_0 \\ \text{with } |I_0| < \infty \\ \text{and } I_0(0) = I(0) \text{ and } I_0 \models \varphi \text{ and } I_0(|I_0|) = I(0)' \\ \text{and } I_0(m)' = I_0(m)'' \text{ for all } m < |I_0| \end{array}$$

2.9 Labels

Definition 29 (*Semantics of labels*)

$$l : \varphi \quad ::= \quad (\neg \mathbf{last} \rightarrow l' \neq 1) \wedge \bullet \square l' = 1 \wedge \exists l. \varphi$$

The purpose of label l is to mark the current program position. Label l is a boolean program variable, its truth value toggles, if and only if the label is satisfied. Thus, satisfaction of the label is independent of its initial value. The label marks the starting transition of φ ; in the first transition – if any – the label is triggered. For the rest of the transitions, the label is not satisfied.

2.10 Restricted expressions

Different types of restricted expressions are considered in the following sections. Here, we define static and dynamic expressions: a static expression must not contain dynamic or program variables and temporal operators, a dynamic expression must not contain primed and double primed dynamic variables as well as program variables or temporal operators.

Definition 30 (*Static and dynamic expressions*) A static expression ε_s is defined to be an expression $\in \mathbf{E}$ which adheres to the following grammar:

$$\varepsilon_s \quad ::= \quad X \mid f(\varepsilon_s, \dots, \varepsilon_s) \mid \varepsilon_s = \varepsilon_s \mid \exists X. \varepsilon_s .$$

Let $v \in \mathbf{X} \cup \mathbf{Y}$, $w \in \mathbf{Y} \cup \mathbf{Z}$. A dynamic expression ε is defined to be an expression $\in \mathbf{E}$ which adheres to the following grammar:

$$\varepsilon ::= X \mid w \mid f(\varepsilon, \dots, \varepsilon) \mid \varepsilon = \varepsilon \mid \exists v. \varepsilon .$$

A boolean dynamic expression $\varepsilon \in \mathbf{E}_{\text{bool}}$ is also called a condition.

A static expression is a special case of a dynamic expression. It is important to note that a dynamic expression can be evaluated in a single state.

Corollary 3 (Properties of dynamic expressions) For a dynamic expression ε , the following property holds:

$$\llbracket \varepsilon \rrbracket_I = \llbracket \varepsilon \rrbracket_{I(0)}$$

Structural induction over dynamic expressions (see Def. 30). □

2.11 Substitution

2.11.1 Free variables

Definition 31 (Free variables) Let $v \in \mathbf{X} \cup \mathbf{Y}$, $w \in \mathbf{Y} \cup \mathbf{Z}$. The function

$$\text{free} : \mathbf{E} \rightarrow \mathbf{X} \cup \mathbf{Y} \cup \mathbf{Z},$$

which calculates for an expression the set of free variables, is defined as follows:

$$\begin{aligned} \text{free}(X) &:= \{X\} \\ \text{free}(w) &:= \{w\} \\ \text{free}(w') &:= \{w\} \\ \text{free}(w'') &:= \{w\} \\ \text{free}(f(e_1, \dots, e_n)) &:= \text{free}(e_1) \cup \dots \cup \text{free}(e_n) \\ \text{free}(e_1 = e_2) &:= \text{free}(e_1) \cup \text{free}(e_2) \\ \text{free}(\exists v. \varphi) &:= \text{free}(\varphi) \setminus \{v\} \\ \text{free}(\exists x. \varphi) &:= \text{free}(\varphi) \setminus \{x\} \\ \text{free}(\varphi_1; \varphi_2) &:= \text{free}(\varphi_1) \cup \text{free}(\varphi_2) \\ \text{free}(\varphi^*) &:= \text{free}(\varphi) \\ \text{free}(\mathbf{step}) &:= \emptyset \\ \text{free}(\varphi_1 \mathbf{until} \varphi_2) &:= \text{free}(\varphi_1) \cup \text{free}(\varphi_2) \\ \text{free}(\lceil x_1, \dots, x_n \rceil) &:= \mathbf{Z} \setminus \{x_1, \dots, x_n\} \end{aligned}$$

$$\begin{aligned}
\text{free}(l_1 :: \varphi \parallel^< l_2 :: \psi) &:= \text{free}(l_1) \cup \text{free}(\varphi) \\
&\quad \cup \text{free}(l_2) \cup \text{free}(\psi) \cup \{\text{blk}\} \\
\text{free}(\varphi_1 \parallel \varphi_2) &:= \text{free}(\varphi_1) \cup \text{free}(\varphi_2) \cup \{\text{blk}\} \\
\text{free}(\{\varphi\}) &:= \text{free}(\varphi) \\
\text{free}(\text{proc}(e_1, \dots, e_n; x_1, \dots, x_m)) &:= \text{free}(e_1) \cup \dots \cup \text{free}(e_n) \cup \mathbf{Z} \\
\text{free}(\langle \varphi \rangle \psi) &:= \text{vars}(\langle \varphi \rangle \psi)
\end{aligned}$$

The definition of free variables is as expected in most cases. However, one of the cases to be discussed is the definition of free variables for frame assumptions. The assumption $\lceil x_1, \dots, x_n \rceil$ is interpreted as an infinite conjunction (see Sect. 2.2.7)

$$\bigwedge_{x \notin \{x_1, \dots, x_n\}} x' = x,$$

therefore $\text{free}(\lceil x_1, \dots, x_n \rceil)$ is defined to be a co-finite set $\mathbf{Z} \setminus \{x_1, \dots, x_n\}$ of program variables. Even more, the derived set of free variables for an assignment $x := e$

$$\begin{aligned}
\text{free}(x := e) &= \text{free}(x' = e \wedge \lceil x \rceil \wedge \circ \text{last}) \\
&= \mathbf{Z} \cup \text{free}(e)
\end{aligned}$$

contains all of the program variables \mathbf{Z} . In other words, as an assignment incorporates a frame assumption, it influences all of the program variables. The same holds true for procedure calls. The semantics of procedures requires all program variables except reference parameters to stutter (see Sect. 2.2.9), whereas the reference parameters can be accessed by the procedure itself. Thus, all of the program variables are affected and are considered free variables of a procedure call. The free variables for some of the more interesting derived program operators of Sect. 2.2.7 are as follows:

$$\begin{aligned}
\text{free}(x := e) &= \mathbf{Z} \cup \text{free}(e) \\
\text{free}(x := ?) &= \mathbf{Z} \setminus \{x\} \\
\text{free}(\text{skip}) &= \mathbf{Z} \\
\text{free}(\text{var } x = e \text{ in } \varphi) &= \{x\} \cup \text{free}(e) \cup \text{free}(\varphi) \\
\text{free}(\text{abort}) &= \mathbf{Z} \\
\text{free}(\text{await } \varphi) &= \mathbf{Z} \cup \text{free}(\varphi) \\
\text{free}(\text{blocked}) &= \{\text{blk}\}
\end{aligned}$$

Overall, program variables can be free even though they do not explicitly occur in an expression. Furthermore, the usability of the coincidence lemma (see below) is limited as all of the program variables can be free variables of an expression.

For a system operator $\langle \varphi \rangle \psi$ the definition of free variables is more relaxed. Function $\text{vars}(e)$ only considers all of the variables which are explicitly mentioned in an expression, neglecting dynamic variables which stutter. Thus the resulting set of variables is finite in all cases. A formal definition for $\text{vars}(e)$ is as follows.

Definition 32 (*System variables*) Let $v \in \mathbf{X} \cup \mathbf{Y}$, $w \in \mathbf{Y} \cup \mathbf{Z}$. For the basic operators, the function

$$\text{vars} : \mathbf{E} \rightarrow \mathbf{X} \cup \mathbf{Y} \cup \mathbf{Z},$$

which calculates the set of system variables, is defined as follows.

$$\begin{aligned} \text{vars}(X) &:= \{X\} \\ \text{vars}(w) &:= \{w\} \\ \text{vars}(w') &:= \{w\} \\ \text{vars}(w'') &:= \{w\} \\ \text{vars}(f(e_1, \dots, e_n)) &:= \text{vars}(e_1) \cup \dots \cup \text{vars}(e_n) \\ \text{vars}(e_1 = e_2) &:= \text{vars}(e_1) \cup \text{vars}(e_2) \\ \text{vars}(\exists v. \varphi) &:= \text{vars}(\varphi) \setminus \{v\} \\ \text{vars}(\exists x. \varphi) &:= \text{vars}(\varphi) \setminus \{x\} \\ \text{vars}(\varphi_1; \varphi_2) &:= \text{vars}(\varphi_1) \cup \text{vars}(\varphi_2) \\ \text{vars}(\varphi^*) &:= \text{vars}(\varphi) \\ \text{vars}(\mathbf{step}) &:= \emptyset \\ \text{vars}(\varphi_1 \mathbf{until} \varphi_2) &:= \text{vars}(\varphi_1) \cup \text{vars}(\varphi_2) \\ \text{vars}(\lceil x_1, \dots, x_n \rceil) &:= \emptyset \\ \text{vars}(l_1 :: \varphi \parallel^< l_2 :: \psi) &:= \text{vars}(l_1) \cup \text{vars}(\varphi) \cup \text{vars}(l_2) \cup \text{vars}(\psi) \\ \text{vars}(\varphi_1 \parallel \varphi_2) &:= \text{vars}(\varphi_1) \cup \text{vars}(\varphi_2) \\ \text{vars}(\{\varphi\}) &:= \text{vars}(\varphi) \\ \text{vars}(\mathit{proc}(e_1, \dots, e_n; x_1, \dots, x_n)) &:= \text{vars}(e_1) \cup \dots \cup \text{vars}(e_n) \cup \{x_1, \dots, x_n\} \\ \text{vars}(\langle \varphi \rangle \psi) &:= \text{vars}(\varphi) \cup \text{vars}(\psi) \end{aligned}$$

Based on the definition of free variables, the coincidence lemma also holds for the introduced temporal logic.

Lemma 2 (*Coincidence lemma*) Let I_1, I_2 be intervals with $|I_1| = |I_2|$. If $I_1(v) = I_2(v)$ for all $v \in \text{free}(e)$ then $\llbracket e \rrbracket_{I_1} = \llbracket e \rrbracket_{I_2}$.

Special case of Theorem 1 (see below). □

As has been mentioned above, the frame assumption affects the usability of the coincidence lemma. The lemma allows for variables which are not free in an expression to be arbitrarily changed. As programs usually refer to all of the program variables, the lemma is of real use only for static and dynamic variables.

However, a frame assumption refers to the program variables in a very restricted manner. It requires the variables to stutter, i.e. the value of the primed variable to be equal to the

value of the unprimed variable $x' = x$, the concrete value of x is of no concern. Therefore, the coincidence lemma can be strengthened as follows.

Theorem 1 (*Strong coincidence lemma*) *Let I_1, I_2 be intervals with $|I_1| = |I_2|$. If*

1. $I_1(v) = I_2(v)$ for all $v \in \text{vars}(e)$
2. $I_1(n)'(x) = I_1(n)(x) \Leftrightarrow I_2(n)'(x) = I_2(n)(x)$ for all $x \in \text{free}(e), n < |I_1|$

i.e. the values of the system variables of e in interval I_1 coincide with the values in I_2 , and the free variables of e stutter in I_1 if and only if they stutter in I_2 , then

$$\llbracket e \rrbracket_{I_1} = \llbracket e \rrbracket_{I_2}$$

See Appendix C.1.3. □

2.11.2 Substitution

In this section, two flavours of substitution are defined. The first definition considers the substitution of logical variables v , the second allows for the (parallel) substitution of program variables x . Due to the definition of frame assumptions, program variables can only be renamed.

Logical variables

Substituting static or dynamic variables (also named logical variables) is as expected. However, static variables can only be replaced by static, dynamic variables only by dynamic expressions. The notion of static and dynamic expressions has been defined in Section 2.10. Here, we define an additional function prm which is necessary to prime the dynamic variables of a dynamic expression.

Definition 33 (*Priming a dynamic expression*) *Let $v \in \mathbf{X} \cup \mathbf{Y}$, $w \in \mathbf{Y} \cup \mathbf{Z}$. Function prm , which is used to prime all variables of a dynamic expression, is defined as follows:*

$$\begin{aligned} \text{prm}(X) &::= X \\ \text{prm}(w) &::= w' \\ \text{prm}(w') &::= w'' \\ \text{prm}(f(\varepsilon_1, \dots, \varepsilon_n)) &::= f(\text{prm}(\varepsilon_1), \dots, \text{prm}(\varepsilon_n)) \\ \text{prm}(\varepsilon_1 = \varepsilon_2) &::= (\text{prm}(\varepsilon_1) = \text{prm}(\varepsilon_2)) \\ \text{prm}(\exists v. \varepsilon) &::= (\exists v. \text{prm}(\varepsilon)) \end{aligned}$$

Although, w' is not a dynamic expression, primed variables are considered as an additional case in the definition above to make function prm applicable twice to a dynamic expression. Thus, the expression can be double primed.

Definition 34 (*Substitution of logical variables*) Let $v \in \mathbf{X} \cup \mathbf{Y}$ and $w \in \mathbf{Y} \cup \mathbf{Z}$. In an expression e , a static variable X can be substituted by a static expression ε_s (denoted as $e^{[\varepsilon_s/X]}$), and a dynamic variable x can be substituted by a dynamic expression ε (denoted as $e^{[\varepsilon/x]}$) as follows:

$$\begin{aligned}
X^{[\varepsilon/v]} &:= \begin{cases} \varepsilon, & \text{if } X \equiv v \\ X, & \text{otherwise} \end{cases} \\
w^{[\varepsilon/v]} &:= \begin{cases} \varepsilon, & \text{if } w \equiv v \\ w, & \text{otherwise} \end{cases} \\
w'^{[\varepsilon/v]} &:= \text{prm}(w^{[\varepsilon/v]}) \\
w''^{[\varepsilon/v]} &:= \text{prm}(\text{prm}(w^{[\varepsilon/v]})) \\
f(e_1, \dots, e_n)^{[\varepsilon/v]} &:= f(e_1^{[\varepsilon/v]}, \dots, e_n^{[\varepsilon/v]}) \\
(e_1 = e_2)^{[\varepsilon/v]} &:= (e_1^{[\varepsilon/v]} = e_2^{[\varepsilon/v]}) \\
(\exists v_0. \varphi)^{[\varepsilon/v]} &:= \begin{cases} (\exists v_0. \varphi), & \text{if } v \equiv v_0 \\ (\exists v_1. \varphi^{[v_1/v_0]}^{[\varepsilon/v]}), & \text{otherwise} \\ v_1 \notin (\text{free}(\varphi) \setminus \{v_0\}) \cup \text{free}(\varepsilon) \end{cases} \\
(\exists x. \varphi)^{[\varepsilon/v]} &:= (\exists x. \varphi^{[\varepsilon/v]}) \\
(\varphi_1; \varphi_2)^{[\varepsilon/v]} &:= (\varphi_1^{[\varepsilon/v]}; \varphi_2^{[\varepsilon/v]}) \\
(\varphi^*)^{[\varepsilon/v]} &:= (\varphi^{[\varepsilon/v]})^* \\
\mathbf{step}^{[\varepsilon/v]} &:= \mathbf{step} \\
(\varphi_1 \mathbf{until} \varphi_2)^{[\varepsilon/v]} &:= (\varphi_1^{[\varepsilon/v]} \mathbf{until} \varphi_2^{[\varepsilon/v]}) \\
[x_1, \dots, x_n]^{[\varepsilon/v]} &:= [x_1, \dots, x_n] \\
(l_1 :: \varphi \parallel^< l_2 :: \psi)^{[\varepsilon/v]} &:= (l_1^{[\varepsilon/v]} :: \varphi^{[\varepsilon/v]} \parallel^< l_2^{[\varepsilon/v]} :: \psi^{[\varepsilon/v]}) \\
(\varphi_1 \parallel \varphi_2)^{[\varepsilon/v]} &:= (\varphi_1^{[\varepsilon/v]} \parallel \varphi_2^{[\varepsilon/v]}) \\
\{\varphi\}^{[\varepsilon/v]} &:= \{\varphi^{[\varepsilon/v]}\} \\
\mathit{proc}(e_1, \dots, e_n; x_1, \dots, x_m)^{[\varepsilon/v]} &:= \mathit{proc}(e_1^{[\varepsilon/v]}, \dots, e_n^{[\varepsilon/v]}; x_1, \dots, x_m) \\
(\langle \varphi \rangle \psi)^{[\varepsilon/v]} &:= (\langle \varphi^{[\varepsilon/v]} \rangle \psi^{[\varepsilon/v]})
\end{aligned}$$

The restriction for the substitution of logical variables can be somehow relaxed. Substituting a static variable X with an arbitrary expression e in formula φ could be defined as follows:

$$\varphi^{[e/X]} \equiv \exists X_0. X_0 = e \wedge \varphi^{[X_0/X]}$$

where $X_0 \notin (\text{free}(\varphi) \setminus \{X\}) \cup \text{free}(e)$. Introducing an additional variable X_0 avoids replacing the static variable X with an expression which evaluates to different values in

different states. The expression e is evaluated once, and the result is stored in X_0 . A similar definition could be proposed for dynamic variables:

$$\varphi[e/x] \equiv \exists x_0. \square (x_0 = e \wedge x'_0 = \text{prm}(e)) \wedge \varphi[x_0/x]$$

There remains, however, the unsolved issue of priming an arbitrary expression $\text{prm}(e)$.

Lemma 3 (*Substituting logical variables*) *Let $v \in \mathbf{X} \cup \mathbf{Y}$.*

1. $\models \varphi \Rightarrow \models \varphi[e/v]$
2. $\models \varphi \Leftrightarrow \models \varphi[e/v]$, if $v \notin \text{free}(\varphi)$
3. $\models \varphi \rightarrow \forall v. \psi \Leftrightarrow \models \varphi \rightarrow \psi[v_0/v]$ where $v_0 \notin \text{free}(\varphi \rightarrow \forall v. \psi)$

See Appendix C.1.4. □

Program variables

For program variables, a very restricted notion of substitution is defined here: program variables can only be substituted by program variables; in other words, they can only be renamed. Here, we define how to simultaneously rename a list of unique “old” variables \vec{x} to a list of unique “new” variables \vec{x}_0 . For example, if

$$\vec{x} := [x_1, x_2, x_3], \quad \vec{x}_0 := [x_3, x_2, x_1],$$

then the variables in \vec{x} and \vec{x}_0 are mutually different (unique), and x_1 would be renamed as x_3 , x_2 would remain unchanged, and x_3 would be renamed as x_1 . We require that the list of new variables \vec{x}_0 is a permutation of \vec{x} as is the case in the example above. The latter requirement avoids the problem of renaming a variable as an existing variable. This is no true restriction, as two lists of old variables \vec{x} and new variables \vec{x}_0 can easily be extended to a permutation: add variables which only occur in \vec{x}_0 to \vec{x} ; rename the variables to variables which only occur in \vec{x} . If

$$\vec{x} := [x_1, x_2], \quad \vec{x}_0 := [x_3, x_2],$$

then x_3 only occurs in \vec{x}_0 ($x_3 \in \vec{x}_0 \setminus \vec{x}$), therefore add x_3 to \vec{x} and rename the variable as x_1 , because this is a variable which only occurs in \vec{x} ($x_1 \in \vec{x} \setminus \vec{x}_0$). The existence of such variables remains without proof here.

Definition 35 (*Renaming of program variables*) *Let $v \in \mathbf{X} \cup \mathbf{Y}$ and $w \in \mathbf{Y} \text{ cup } \mathbf{Z}$. Let \vec{x} be a list of unique program variables. Let \vec{x}_0 be a permutation of \vec{x} . Then renaming the variables of \vec{x} as \vec{x}_0 in expression e (denoted as $(e)_{\vec{x}}^{\vec{x}_0}$) is defined as follows:*

$$(X)_{\vec{x}}^{\vec{x}_0} := X$$

$$\begin{aligned}
(w)_{\vec{x}}^{\vec{x}_0} &:= \begin{cases} \vec{x}_0(i), & \text{if } w \equiv \vec{x}(i) \\ w, & \text{otherwise} \end{cases} \\
(w')_{\vec{x}}^{\vec{x}_0} &:= ((w)_{\vec{x}}^{\vec{x}_0})' \\
(w'')_{\vec{x}}^{\vec{x}_0} &:= ((w)_{\vec{x}}^{\vec{x}_0})'' \\
(f(\vec{e}))_{\vec{x}}^{\vec{x}_0} &:= f((\vec{e})_{\vec{x}}^{\vec{x}_0}) \\
(e_1 = e_2)_{\vec{x}}^{\vec{x}_0} &:= (e_1)_{\vec{x}}^{\vec{x}_0} = (e_2)_{\vec{x}}^{\vec{x}_0} \\
(\exists v. \varphi)_{\vec{x}}^{\vec{x}_0} &:= \exists v. (\varphi)_{\vec{x}}^{\vec{x}_0} \\
(\exists x. \varphi)_{\vec{x}}^{\vec{x}_0} &:= \exists (x)_{\vec{x}}^{\vec{x}_0}. (\varphi)_{\vec{x}}^{\vec{x}_0} \\
(\varphi_1; \varphi_2)_{\vec{x}}^{\vec{x}_0} &:= (\varphi_1)_{\vec{x}}^{\vec{x}_0}; (\varphi_2)_{\vec{x}}^{\vec{x}_0} \\
(\varphi^*)_{\vec{x}}^{\vec{x}_0} &:= (\varphi)_{\vec{x}}^{\vec{x}_0} * \\
(\text{step})_{\vec{x}}^{\vec{x}_0} &:= \text{step} \\
(\varphi_1 \text{ until } \varphi_2)_{\vec{x}}^{\vec{x}_0} &:= (\varphi_1)_{\vec{x}}^{\vec{x}_0} \text{ until } (\varphi_2)_{\vec{x}}^{\vec{x}_0} \\
([\mathbf{x}_1, \dots, \mathbf{x}_n])_{\vec{x}}^{\vec{x}_0} &:= [(\mathbf{x}_1)_{\vec{x}}^{\vec{x}_0}, \dots, (\mathbf{x}_n)_{\vec{x}}^{\vec{x}_0}] \\
(l_1 :: \varphi \parallel^< l_2 :: \psi)_{\vec{x}}^{\vec{x}_0} &:= (l_1)_{\vec{x}}^{\vec{x}_0} :: (\varphi)_{\vec{x}}^{\vec{x}_0} \parallel^< (l_2)_{\vec{x}}^{\vec{x}_0} :: (\psi)_{\vec{x}}^{\vec{x}_0} \\
(\varphi_1 \parallel \varphi_2)_{\vec{x}}^{\vec{x}_0} &:= (\varphi_1)_{\vec{x}}^{\vec{x}_0} \parallel (\varphi_2)_{\vec{x}}^{\vec{x}_0} \\
(\{\varphi\})_{\vec{x}}^{\vec{x}_0} &:= \{(\varphi)_{\vec{x}}^{\vec{x}_0}\} \\
(\text{proc}(e_1, \dots, e_n; \mathbf{x}_1, \dots, \mathbf{x}_n))_{\vec{x}}^{\vec{x}_0} &:= \text{proc}((e_1)_{\vec{x}}^{\vec{x}_0}, \dots, (e_n)_{\vec{x}}^{\vec{x}_0}; \\
&\quad (\mathbf{x}_1)_{\vec{x}}^{\vec{x}_0}, \dots, (\mathbf{x}_n)_{\vec{x}}^{\vec{x}_0}) \\
(\langle \langle \varphi \rangle \psi \rangle)_{\vec{x}}^{\vec{x}_0} &:= (\langle (\varphi)_{\vec{x}}^{\vec{x}_0} \rangle (\psi)_{\vec{x}}^{\vec{x}_0})
\end{aligned}$$

Example Consider the binom example of Section 2.1. For the implementation of Binom, we would like to rename the result variable b as r and the semaphore r as s . In order to satisfy the permutation requirement, we also rename s as b .

$$\left(\begin{array}{l} \text{var } n = n, k = k \text{ in} \\ \text{var } y_1 = n, y_2 = 1, t_1 = ?, t_2 = ?, r = 1 \text{ in} \\ b := 1; \\ \text{while } y_1 > n - k \text{ do} \parallel \text{while } y_2 \leq k \text{ do do} \\ \quad P(; r); \quad \text{await } y_1 + y_2 \leq n \\ \quad t_1 := b * y_1; \quad P(; r); \\ \quad b := t_1; \quad t_2 := b \text{ div } y_2; \\ \quad V(; r); \quad b := t_2; \\ \quad y_1 := y_1 - 1 \quad V(; r); \\ \quad \quad y_2 := y_2 + 1 \end{array} \right) \begin{array}{l} [r,s,b] \\ [b,r,s] \end{array}$$

results in

```

var n = n, k = k in
  var y1 = n, y2 = 1, t1 = ?, t2 = ?, s = 1 in
    r := 1;
    while y1 > n - k do || while y2 ≤ k do do
      P(; s);                await y1 + y2 ≤ n
      t1 := r * y1;          P(; s);
      r := t1;              t2 := r div y2;
      V(; s);                r := t2;
      y1 := y1 - 1          V(; s);
                          y2 := y2 + 1
  
```

Lemma 4 (*Renaming program variables*) *Let \vec{x} be a list of unique program variables. Let \vec{x}_0 be a permutation of \vec{x} . The following property holds for the renaming of program variables.*

$$\models \varphi \Leftrightarrow \models (\varphi)_{\vec{x}}^{\vec{x}_0}$$

We sketch the proof here. A more detailed proof can be found in Appendix C.1.5.

For an arbitrary I , construct $I_{\vec{x}}^{\vec{x}_0}$ with

$$\begin{aligned}
 |I_{\vec{x}}^{\vec{x}_0}| &= |I| \\
 I_{\vec{x}}^{\vec{x}_0}(X) &= I(X) \\
 I_{\vec{x}}^{\vec{x}_0}(x) &= I(x) \\
 I_{\vec{x}}^{\vec{x}_0}(x) &= \begin{cases} I(\vec{x}_0(i)), & \text{if } x \equiv \vec{x}(i) \\ I(x), & \text{otherwise} \end{cases}
 \end{aligned}$$

(Because \vec{x}_0 is a permutation of \vec{x} , interval $I_{\vec{x}}^{\vec{x}_0}$ is uniquely defined.) Show that for arbitrary I

$$\llbracket (e)_{\vec{x}}^{\vec{x}_0} \rrbracket_I = \llbracket e \rrbracket_{I_{\vec{x}}^{\vec{x}_0}} .$$

This is established with structural induction over expression e . □

2.12 Conclusion

We have defined a logic where operators for parallel programs and temporal formulas are semantically the same. This approach is motivated by Interval Temporal Logic. Using *double primed variables*, we were able to define interleaving of arbitrary temporal formulas. We explicitly consider the environment on a semantics level: after each system transition, an environment transition is taken. The alternation of system and environment transitions is somehow related to the concept of stuttering steps of TLA. The relation between the

two approaches must be further examined. *In this approach, the separation of system and environment transitions is the key to the definition of a compositional semantics of interleaving.*

The semantics is such that not only interleaving but all of the operators are *compositional*, i.e. sub formulas can be replaced with more abstract formulas. This is elaborated in Chapter 7.

We have chosen to explicitly define *blocked transitions*; if the special variable `blk` toggles, then the transition is blocked. Compared to other approaches, we are able to verify that the system does not deadlock. Furthermore, a pure parallel program with program constructs only (including **await**) is consistent, i.e. the set of traces is nonempty. We, however, receive a more complicated semantics for interleaving and nondeterministic choice.

The use of SOS rules to give an operational semantics to certain operators must be further elaborated. In our opinion, SOS rules give a more intuitive semantics for interleaving and nondeterministic choice. Our integration of operational and declarative semantics still is an ad hoc solution.

Chapter 3

Calculus

Our proof strategy is to *symbolically execute* temporal formulas, parallel programs being just a special case thereof (see Chapter 4). Formulas are transformed to normal form (see Section 4.2) which separates the possible first transitions and the corresponding temporal formulas describing the system in the next state. In general, a normal form for interleaved temporal formulas can only be achieved, if rules are applied to sub-formulas. This has been discussed in Section 1.5.6. For this reason, the calculus is based on *rewriting*. The following sections introduce our strategy of rewriting, where special care has been taken to preserve context information while descending into a sub-formula. For this purpose, *congruence rules* are defined; this approach is motivated by the theorem prover Isabelle [25, 17].

In order to avoid exponential growth of proof trees, the premises where execution results in the same system configuration are combined. This is what we call *sequencing* (see Chapter 5). It effectively avoids exponential growth in practice and can be applied independent from the temporal property under verification. If execution loops, the proof requires an inductive argument. Rules for *induction* are discussed in Chapter 6. The different rules are based on a simple induction principle which is powerful enough to establish safety and liveness properties. As operators are compositional, *abstraction* can be used to decompose large proofs (see Chapter 7).

Considerable effort has been spent to ensure that the calculus rules can be automatically applied to a large extent. Almost all of the rules are invertible ensuring that, if the conclusion is provable, the resulting premises remain valid. The overall strategy is

- to symbolically execute a given proof obligation,
- to simplify the PL formulas describing the current state,
- to combine premises with the same system configuration, and
- to use induction, if a system loop has been executed.

Completeness of the set of rules has not been examined. The calculus has been refined with applications, and therefore, the rules are most advanced for the operators which have regularly been encountered in practice. Appendix B lists the current set of rules, which are also implemented in the KIV interactive verification environment.

For the symbolic execution of interleaved programs, the sub-programs must be executed, i.e., rewritten in normal form, in advance to the execution of the top level interleaving operator (see Section 1.5.6). This requires the application of rules to sub-formulas. Furthermore, our proof method must also consider generalisation and sequencing and must be automatable to a large extent. This leads to the following requirements for our proof method:

1. It must be possible to rewrite arbitrary sub-expressions. This is necessary for the execution of sub-programs.
2. Under certain conditions, it must be possible to weaken or strengthen sub-formulas. This is necessary to generalise conditions of sub-programs for induction.
3. It must be possible to not only split cases, but also to unite cases. This is necessary to implement the concept of sequencing (see Chapter 5).
4. The approach should be automatable to a large extent. Thus, which rule to apply next must be obvious in most cases.

Section 3.1 discusses the suitability of existing approaches. Our approach in Sect. 3.2 is a variant of these which satisfies the requirements above.

3.1 Existing approaches

3.1.1 Sequent Calculus

KIV is based on a sequent calculus. A sequent calculus uses sequents

$$\Gamma \vdash \Delta$$

with a list of formulas $\Gamma \equiv \varphi_1, \dots, \varphi_n$ which is called the antecedent and a list of formulas $\Delta \equiv \psi_1, \dots, \psi_m$ which is called the succedent. The sequent is interpreted as

$$\text{Cl}_\forall(\varphi_1 \wedge \dots \wedge \varphi_n \rightarrow \psi_1 \vee \dots \vee \psi_m)$$

i.e., the conjunction of the formulas in the antecedent imply the disjunction of the succedent. All free variables of the resulting formula are universally quantified.

Sequent rules are used to construct proofs. A sequent rule

$$\frac{P_1 \dots P_n}{C} \text{ name, } \quad \text{if cond}$$

$$\begin{array}{c}
\frac{\varphi, \Gamma \vdash_{\text{DL}} \Delta}{\langle \text{skip} \rangle \varphi, \Gamma \vdash_{\text{DL}} \Delta} \text{skip}_l \qquad \frac{\Gamma \vdash_{\text{DL}} \varphi, \Delta}{\Gamma \vdash_{\text{DL}} \langle \text{skip} \rangle \varphi, \Delta} \text{skip}_r \\
\\
\frac{}{\langle \text{abort} \rangle \varphi, \Gamma \vdash_{\text{DL}} \Delta} \text{abort}_l \qquad \frac{\Gamma \vdash_{\text{DL}} \Delta}{\Gamma \vdash_{\text{DL}} \langle \text{abort} \rangle \varphi, \Delta} \text{abort}_r \\
\\
\frac{\varphi[x_0/x], x_0 = e, \Gamma \vdash_{\text{DL}} \Delta}{\langle x := e \rangle \varphi, \Gamma \vdash_{\text{DL}} \Delta} \text{asg}_l \qquad \frac{x_0 = e, \Gamma \vdash_{\text{DL}} \varphi[x_0/x], \Delta}{\Gamma \vdash_{\text{DL}} \langle x := e \rangle \varphi, \Delta} \text{asg}_r \\
\\
\frac{\langle \alpha \rangle \langle \beta \rangle \varphi, \Gamma \vdash_{\text{DL}} \Delta}{\langle \alpha; \beta \rangle \varphi, \Gamma \vdash_{\text{DL}} \Delta} \text{comp}_l \qquad \frac{\Gamma \vdash_{\text{DL}} \langle \alpha \rangle \langle \beta \rangle \varphi, \Delta}{\Gamma \vdash_{\text{DL}} \langle \alpha; \beta \rangle \varphi, \Delta} \text{comp}_r \\
\\
\frac{\varepsilon, \langle \alpha \rangle \varphi, \Gamma \vdash_{\text{DL}} \Delta \quad \langle \beta \rangle \varphi, \Gamma \vdash_{\text{DL}} \varepsilon, \Delta}{\langle \text{if } \varepsilon \text{ then } \alpha \text{ else } \beta \rangle \varphi, \Gamma \vdash_{\text{DL}} \Delta} \text{if}_l \\
\\
\frac{\varepsilon, \Gamma \vdash_{\text{DL}} \langle \alpha \rangle \varphi, \Delta \quad \Gamma \vdash_{\text{DL}} \varepsilon, \langle \beta \rangle \varphi, \Delta}{\Gamma \vdash_{\text{DL}} \langle \text{if } \varepsilon \text{ then } \alpha \text{ else } \beta \rangle \varphi, \Delta} \text{if}_r \\
\\
\frac{\exists n. \langle \text{loop if } \varepsilon \text{ then } \alpha \text{ else skip times } n \rangle (\neg \varepsilon \wedge \varphi), \Gamma \vdash_{\text{DL}} \Delta}{\langle \text{while } \varepsilon \text{ do } \alpha \rangle \varphi, \Gamma \vdash_{\text{DL}} \Delta} \text{while}_l \\
\\
\frac{\langle \text{skip} \rangle \varphi, \Gamma \vdash_{\text{DL}} \Delta}{\langle \text{loop } \alpha \text{ times } 0 \rangle \varphi, \Gamma \vdash_{\text{DL}} \Delta} \text{loop}_l z \\
\\
\frac{\langle \alpha; \text{loop } \alpha \text{ times } n \rangle \varphi, \Gamma \vdash_{\text{DL}} \Delta}{\langle \text{loop } \alpha \text{ times } n + 1 \rangle \varphi, \Gamma \vdash_{\text{DL}} \Delta} \text{loop}_l s
\end{array}$$

Figure 3.1: Sequent rules for Dynamic Logic [13]

consists of a sequent C which is called *conclusion* and sequents P_1, \dots, P_n which are called *premises*. To apply the rule, the side condition *cond* must also be satisfied. A rule without premises is an axiom.

A derivation is a successive application of sequent rules which gives a proof tree.

3.1.2 Dynamic Logic

Dynamic Logic (DL) [11] [13] defines formulas $[\alpha] \varphi$ (“box $\alpha \varphi$ ”) and $\langle \alpha \rangle \varphi$ (“diamond $\alpha \varphi$ ”) where α is a sequential program and φ a DL formula. Formula $[\alpha] \varphi$ states that if program α terminates, then property φ holds after program execution. Formula $\langle \alpha \rangle \varphi$ states that program α terminates and property φ finally holds.

The sequent calculus of [13] (see Figure 3.1) can be used to symbolically execute programs. Note that the leading operator of programs determines the applicable proof rules.

Therefore, execution is automatic to such a large extent that it satisfies requirement 4. Lemmas can be defined to establish several cases with one proof. However, for parallel programs it is necessary to regularly unite cases, which renders the lemma mechanism too complicated. Thus, requirement 3 is only partially satisfied. For the verification of sequential programs, it is sufficient to apply rules to the top level formulas of the antecedent and succedent. For parallel programs the approach must be enhanced with respect to requirements 1 and 2. The first requirement is best satisfied with rewriting.

3.1.3 Rewriting

In the literature, rewriting is used to apply rules to sub-expressions. A rewrite rule

$$\textit{name: } \varphi \rightarrow (e_1 = e_2)$$

states that expression e_1 can be replaced by an equal expression e_2 if the precondition φ holds. In a sequent calculus, rewrite rules can be applied as follows:

$$\frac{\vdash \varphi \rightarrow e_1 = e_2 \quad \Gamma \vdash \varphi, \Delta \quad \Gamma[e_2/e_1] \vdash \Delta[e_2/e_1]}{\Gamma \vdash \Delta} \textit{insert rewrite lemma}$$

where $\Gamma[e_2/e_1]$ replaces all “free occurrences” of e_1 with e_2 in Γ . Replacement of expressions is not straightforward to define, especially quantifiers raise difficulties.

Example Consider the following rewrite rule

$$\textit{dec-inc: } 0 < n \rightarrow (n - 1 + 1 = n)$$

which we call *dec-inc*. It can be used to simplify the successive application of decrementation and incrementation of natural numbers. If we use rule *insert rewrite lemma* to apply the rewrite rule to the sequent

$$0 < n \vdash (n - 1 + 1) + m \geq m$$

we receive

$$0 < n \vdash n + m \geq m$$

as third premise. The sub-expression has been replaced.

Rule *insert rewrite lemma* requires the precondition to hold for all cases. A more general approach evaluates the precondition in the context of the expression to be rewritten. In the Isabelle theorem prover [24], so called *congruence rules* are used to define the context of a sub-expression. The congruence rule

$$\frac{\vdash \varphi_1 = \varphi_2 \quad \neg \varphi_2 \vdash \psi_1 = \psi_2}{\vdash (\varphi_1 \vee \psi_1) = (\varphi_2 \vee \psi_2)} \textit{dis cong}$$

which we call *dis cong* expresses that a disjunction $\varphi_1 \vee \psi_1$ can be rewritten to $\varphi_2 \vee \psi_2$ if φ_1 is rewritten to φ_2 and ψ_1 to ψ_2 . For rewriting ψ_1 , an additional assumption $\neg \varphi_2$ can be used.

Example Consider the disjunction

$$\vdash n = 0 \vee (n - 1 + 1) + m \geq m$$

If we apply the congruence rule *dis cong* with $\varphi_2 \equiv n = 0$ and $\psi_2 \equiv n + m \geq m$, we receive

$$\frac{\vdash (n = 0) = (n = 0) \quad n \neq 0 \vdash ((n - 1 + 1) + m \geq m) = (n + m \geq m)}{\vdash (n = 0 \vee (n - 1 + 1) + m \geq m) = (n = 0 \vee n + m \geq m)}$$

The first premise is trivial; for the second premise, the additional precondition $n \neq 0$ can be used to establish $0 < n$ and to replace $n - 1 + 1$ by n .

Sophisticated simplification strategies can be defined which make use of the context to automatically rewrite an expression to a “simpler” equivalent expression.

Rewriting especially satisfies requirement 1. Arbitrary sub-expressions can be rewritten. However, sub-formulas can only be rewritten to equivalent sub-formulas, therefore, requirement 2 is not fulfilled.

3.2 Rewriting with context

Our approach differs in two aspects from rewriting as described above. First, besides automatic simplification, we would like to interactively apply rewrite rules to sub-expressions. Second, we would like to “abstract” arbitrary sub-formulas to more general (“weaker” or “stronger”) formulas.

3.2.1 Context sequent

For our purposes, we shall define a restricted form of a sequent with a single formula in the succedent. The formula presents the focus for our proof effort.

Definition 36 (*Sequent*) *Let a list of formulas Γ and a formula φ be given. We define a sequent*

$$\Gamma \vdash \varphi$$

to consist of an antecedent Γ and succedent φ . We shall refer to formulas in Γ as preconditions and call φ the proof obligation.

A sequent calculus is used to establish φ . Proof rules need not be invertible, e.g., the proof rule

$$\frac{\Gamma \vdash \psi \rightarrow \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi} \text{ mp}$$

is sound: if $\psi \rightarrow \varphi$ (premise 1), it is sufficient to prove ψ (premise 2) in order to establish φ (conclusion). Formula ψ can be stronger than the original proof obligation.

For our general purpose of rewriting sub-expressions, we shall define sequents with an additional “proof mode”. In one mode, it is required to weaken instead of strengthen a formula, in another, formulas must be replaced by equivalent formulas only. The latter mode will be defined such that not only formulas but arbitrary expressions can be rewritten.

Definition 37 (*Context sequent*) Let two expressions e_1 and e_2 and a list of formulas Γ be given. The sequent

$$\Gamma \vdash_{\mathbf{c}} e_1 = e_2$$

is called a context sequent with context Γ , context expression e_1 , and resulting expression e_2 . For formulas φ_1 and φ_2 , the sequent

$$\Gamma \vdash_{\mathbf{c}} \varphi_1 \subset \varphi_2$$

is called a negative context sequent, and

$$\Gamma \vdash_{\mathbf{c}} \varphi_1 \supset \varphi_2$$

a positive context sequent. Formula φ_1 is the context formula and φ_2 the resulting formula.

Expression e_1 (resp. formula φ_1) is to be rewritten, with e_2 (resp. φ_2) being the expected result. If the succedent is an equation, the proof must establish the equality of the expressions. Otherwise, if the operator in the succedent is \supset (resp. \subset), formula φ_2 must be stronger (resp. weaker) than φ_1 . The operator $=$, \subset , or \supset in the succedent is interpreted as the current proof mode. Operators \subset and \supset are used instead of a simple implication \rightarrow to ensure that the formula to be rewritten is always on the left.

Definition 38 (*Semantics of sequents*) With $\Gamma \equiv \psi_1, \dots, \psi_n$, the semantics of sequents is as follows:

$$\begin{aligned} \Gamma \vdash \varphi & \text{ iff } \models \psi_1 \wedge \dots \wedge \psi_n \rightarrow \varphi \\ \Gamma \vdash_{\mathbf{c}} e_1 = e_2 & \text{ iff } \models \psi_1 \wedge \dots \wedge \psi_n \rightarrow e_1 = e_2 \\ \Gamma \vdash_{\mathbf{c}} \varphi_1 \subset \varphi_2 & \text{ iff } \models \psi_1 \wedge \dots \wedge \psi_n \rightarrow (\varphi_1 \rightarrow \varphi_2) \\ \Gamma \vdash_{\mathbf{c}} \varphi_1 \supset \varphi_2 & \text{ iff } \models \psi_1 \wedge \dots \wedge \psi_n \rightarrow (\varphi_2 \rightarrow \varphi_1) \end{aligned}$$

3.2.2 Basic rules

For sequents $\Gamma \vdash \varphi$, four simple rules are sufficient, the main proof being constructed with rules for rewriting context sequents (see below).

$$\frac{}{\Gamma \vdash \text{true}} \text{true} \quad \frac{}{\Gamma_1, \varphi, \Gamma_2 \vdash \varphi} \text{ax}$$

$$\frac{\Gamma \vdash \varphi_1 \quad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2} \text{con} \quad \frac{\Gamma \vdash \varphi[v_0/v]}{\Gamma \vdash \forall v. \varphi} \text{all}$$

where $v_0 \in \mathbf{Y}$ is fresh with respect to $(\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(\Gamma)$. The first rule finishes a proof after the proof obligation has been rewritten to true. The second rule makes use of a precondition to establish φ . The last two rules are used to split conjunctions into premises and to eliminate quantifiers.

3.2.3 Rewriting

Rewriting is initiated as follows:

$$\frac{\Gamma \vdash_{\mathbf{c}} \varphi \supset \psi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi} \text{rewrite}$$

If φ can be rewritten to ψ (first premise), it is sufficient to prove ψ (second premise) to establish φ (conclusion). Formula ψ must be equivalent or stronger than φ ; therefore, the initial proof mode is \supset .

In order to apply rule *rewrite*, it is necessary to provide the result ψ of rewriting φ . In practice, we shall leave the value of ψ unknown (we shall use “meta variables”) and will insert the result for ψ , after rewriting of φ is finished. To finish rewriting, we use rules

$$\frac{}{\Gamma \vdash_{\mathbf{c}} \varphi \supset \varphi} \text{close}^{\supset} \quad \frac{}{\Gamma \vdash_{\mathbf{c}} \varphi \subset \varphi} \text{close}^{\subset} \quad \frac{}{\Gamma \vdash_{\mathbf{c}} e = e} \text{close}^{\text{=}}$$

which effectively take the current formula as the result. For an example, see below.

3.2.4 Rewrite rules

For proof mode \supset , rewrite rules adhere to the following pattern

$$\frac{}{\Gamma \vdash_{\mathbf{c}} \varphi_1 \supset \varphi_2} \langle \text{name} \rangle^{\supset}$$

A context formula φ_1 is rewritten to φ_2 . Context Γ remains unchanged.

Example For operator \vee , the following rule

$$\frac{}{\Gamma \vdash_{\mathbf{c}} \varphi \vee \text{true} \supset \text{true}} \text{dis true } \mathcal{D}^{\supset}$$

holds. For arbitrary φ , a formula $\varphi \vee \text{true}$ can be rewritten to true . The rule is applicable for proof mode \supset , and for any context Γ .

This rewrite rule is applied to the proof obligation $\vdash n = 0 \vee \text{true}$ as follows: we apply *rewrite* to initiate rewriting.

$$\frac{\vdash_{\mathbf{c}} n = 0 \vee \text{true} \supset \psi \quad \vdash \psi}{\vdash n = 0 \vee \text{true}} \text{rewrite}$$

The first premise matches the conclusion of *dis true* 2^{\supset} . After applying the rule, we receive

$$\frac{\frac{\vdash_{\mathbf{c}} n = 0 \vee \text{true} \supset \text{true}}{\vdash n = 0 \vee \text{true}} \text{dis true } 2^{\supset} \quad \vdash \text{true}}{\vdash n = 0 \vee \text{true}} \text{rewrite}$$

Formula ψ has been instantiated with true ; the original formula has been rewritten. We can now apply rule *true* to finish the proof.

$$\frac{\vdash \text{true}}{\vdash n = 0 \vee \text{true}} \text{true} \text{rewrite}$$

In general, we use so called rewrite lemmas to rewrite expressions. In the simplest case, a rewrite lemma is an equivalence or an equation. Simple rewrite rules to apply the lemma in different proof modes are defined as follows.

Definition 39 (*Simple rewrite rules*) Let

$$\begin{aligned} \text{lem}_1: \quad \varphi_1 &\leftrightarrow \varphi_2 \\ \text{lem}_2: \quad e_1 &= e_2 \end{aligned}$$

We define simple rewrite rules rw^* as follows:

$$\frac{\Gamma \vdash \text{lem}_1}{\Gamma \vdash_{\mathbf{c}} \varphi_1 \supset \varphi_2} \text{rw}^{\supset} \quad \frac{\Gamma \vdash \text{lem}_1}{\Gamma \vdash_{\mathbf{c}} \varphi_1 \subset \varphi_2} \text{rw}^{\subset} \quad \frac{\Gamma \vdash \text{lem}_2}{\Gamma \vdash_{\mathbf{c}} e_1 = e_2} \text{rw}^{\text{=}}$$

The first premise of a simple rewrite rule is normally established by a lemma or axiom and can therefore be closed with rule *ax*. In this case, instead of writing

$$\frac{\overline{\Gamma \vdash \text{lem}} \text{ax}}{\Gamma \vdash_{\mathbf{c}} \varphi_1 \supset \varphi_2} \text{rw}^{\supset}$$

we shall often simply write

$$\overline{\Gamma \vdash_{\mathbf{c}} \varphi_1 \supset \varphi_2} \text{lem}^{\supset}$$

Example Instead of the rule from the example above, we can define an axiom

$$\text{dis true 2: } \varphi_1 \vee \text{true} \leftrightarrow \text{true}$$

and use rule rw^\supset to apply the axiom as follows:

$$\frac{\frac{}{\vdash n = 0 \vee \text{true} \leftrightarrow \text{true}}{ax}}{\vdash_{\mathbf{c}} n = 0 \vee \text{true} \supset \text{true}}{rw^\supset}$$

In short, we write

$$\frac{}{\vdash_{\mathbf{c}} n = 0 \vee \text{true} \supset \text{true}} \text{dis true 2}^\supset$$

Discussion: The example of this section may give the impression that our approach is very complicated to achieve simple rewriting. Bear in mind, however, that we aim at a more general target.

3.2.5 Conditional rewrite rules

Rewriting an expression e_1 to e_2 often requires an additional precondition ψ . A conditional rewrite rule contains an additional premise to establish ψ .

Definition 40 (*Conditional rewrite rules*) Let

$$\begin{aligned} \text{lem}_1: & \quad \psi \rightarrow (\varphi_1 \leftrightarrow \varphi_2) \\ \text{lem}_2: & \quad \psi \rightarrow (e_1 = e_2) \end{aligned}$$

We define conditional rewrite rules $rwpre^*$ as follows:

$$\begin{aligned} \frac{\Gamma \vdash \text{lem}_1 \quad \Gamma \vdash \psi}{\Gamma \vdash_{\mathbf{c}} \varphi_1 \supset \varphi_2} \text{rwpre}^\supset & \quad \frac{\Gamma \vdash \text{lem}_1 \quad \Gamma \vdash \psi}{\Gamma \vdash_{\mathbf{c}} \varphi_1 \subset \varphi_2} \text{rwpre}^\subset \\ \frac{\Gamma \vdash \text{lem}_2 \quad \Gamma \vdash \psi}{\Gamma \vdash_{\mathbf{c}} e_1 = e_2} \text{rwpre}^= & \end{aligned}$$

The second premise of a conditional rewrite rule requires ψ to be derived from the current context Γ .

3.2.6 Congruence rules

In addition to rewriting the top level formula, we would like to apply rules to sub-expressions. For this, we use so called *congruence rules*. In the following definition, $f(e)$ denotes an expression f which contains a sub-expression e . Note that formulas are expressions of sort `bool`.

Definition 41 (*Simple congruence rules*) Let

$$\text{lem: } e_1 = e_2 \rightarrow (f(e_1) = f(e_2))$$

We define simple congruence rules for every proof mode $*$ \in $\{\supset, \subset, =\}$ as follows:

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma \vdash_{\mathbf{c}} e_1 = e_2}{\Gamma \vdash_{\mathbf{c}} f(e_1) * f(e_2)} \text{cong}^*$$

In the second premise, the sub-expression e_1 has been lifted; e_1 is the context expression of the context sequent and can be rewritten. The result e_2 of rewriting e_1 replaces the original sub-expression in the resulting expression of the conclusion. A sub-expression e_1 can be replaced by an equivalent expression e_2 in every proof mode.

Sub-formulas can also be replaced by stronger or weaker formulas using the following congruence rules.

Definition 42 (*Congruence rules with implication*) Let

$$\text{lem: } (\varphi_1 \rightarrow \varphi_2) \rightarrow (\psi(\varphi_1) \rightarrow \psi(\varphi_2))$$

We define congruence rules with implication as follows:

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma \vdash_{\mathbf{c}} \varphi_2 \supset \varphi_1}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \psi(\varphi_1)} \text{congimp}^{\supset} \quad \frac{\Gamma \vdash \text{lem} \quad \Gamma \vdash_{\mathbf{c}} \varphi_1 \subset \varphi_2}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \subset \psi(\varphi_2)} \text{congimp}^{\subset}$$

Lemma *lem* establishes $\psi(\varphi_1) \rightarrow \psi(\varphi_2)$, if $\varphi_1 \rightarrow \varphi_2$. Note that formula $\psi(\varphi_2)$ can be weaker than $\psi(\varphi_1)$. Therefore, if proof mode is positive (rule congimp^{\supset}), the weaker formula $\psi(\varphi_2)$ in the conclusion is replaced by the stronger formula $\psi(\varphi_1)$ in the third premise, and vice versa in negative proof mode (rule congimp^{\subset}).

Example For the operator \wedge , the following axiom is sound.

$$\text{con lem 1: } (\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1 \wedge \psi \rightarrow \varphi_2 \wedge \psi)$$

This axiom can be used to rewrite the first sub-formula of a conjunction. Consider the proof obligation

$$\vdash (n = 0 \vee \text{true}) \wedge n = 1$$

Again, we start with rule *rewrite* to receive

$$\frac{\vdash_{\mathbf{c}} (n = 0 \vee \text{true}) \supset \psi \quad \vdash \psi}{\vdash (n = 0 \vee \text{true}) \wedge n = 1} \text{rewrite}$$

With rule $congimp^\supset$, we lift the first sub-formula of the conjunction, obtaining

$$\frac{\frac{\vdash \text{con lem } 1' \quad n = 1 \vdash_{\mathbf{c}} n = 0 \vee \text{true} \supset \psi}{\vdash_{\mathbf{c}} (n = 0 \vee \text{true}) \wedge n = 1 \supset \psi \wedge n = 1} \quad congimp^\supset}{\vdots} \quad \vdots$$

The first premise is an axiom and can be closed with rule ax . In the conclusion, the original result ψ has been partially instantiated and now reads $\psi \wedge n = 1$. In the second premise, formula $n = 0 \vee \text{true}$ is the top level formula and can be rewritten with rule $dis \text{ true } \mathcal{Q}^\supset$.

$$\frac{\frac{\vdots \quad \frac{\vdash_{\mathbf{c}} n = 0 \vee \text{true} \supset \text{true}}{\vdash_{\mathbf{c}} (n = 0 \vee \text{true}) \wedge n = 1 \supset \text{true} \wedge n = 1} \quad dis \text{ true } \mathcal{Q}^\supset}{\vdash_{\mathbf{c}} (n = 0 \vee \text{true}) \wedge n = 1 \supset \text{true} \wedge n = 1} \quad congimp^\supset}{\vdash (n = 0 \vee \text{true}) \wedge n = 1} \quad \vdash \text{true} \wedge n = 1 \quad \text{rewrite}$$

Application of rule $dis \text{ true } \mathcal{Q}^\supset$ instantiates ψ with true . Thus, the second premise of $rewrite$ now reads $\vdash \text{true} \wedge n = 1$ which differs from the conclusion in that the first sub-formula of the conjunction has been rewritten.

The first premise of rules $cong^*$ is often established using rule ax . Therefore, instead of writing

$$\frac{\frac{\Gamma \vdash \text{lem}}{\Gamma \vdash_{\mathbf{c}} \varphi_2 \supset \varphi_1} \quad ax}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \psi(\varphi_1)} \quad cong^\supset$$

we shall often write

$$\frac{\Gamma \vdash_{\mathbf{c}} \varphi_2 \supset \varphi_1}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \psi(\varphi_1)} \quad lem^\supset$$

3.2.7 Congruence rules with additional context

For many logical operators, additional preconditions can be assumed while rewriting a sub-formula.

Definition 43 (*Congruence rules with additional context*) *Let*

$$\begin{aligned} lem_1: & \quad (\varphi \rightarrow e_1 = e_2) \rightarrow (f(e_1) = f(e_2)) \\ lem_2: & \quad (\varphi \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow (\psi(\varphi_1) \rightarrow \psi(\varphi_2)) \end{aligned}$$

We define congruence rules with additional context as follows:

$$\frac{\Gamma \vdash lem_1 \quad \Gamma, \text{norm}(\varphi) \vdash_{\mathbf{c}} e_1 = e_2}{\Gamma \vdash_{\mathbf{c}} f(e_1) * f(e_2)} \text{congpre}^*$$

$$\frac{\Gamma \vdash lem_2 \quad \Gamma, \text{norm}(\varphi) \vdash \varphi_2 \supset \varphi_1}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \psi(\varphi_1)} \text{congimppre}^{\supset}$$

$$\frac{\Gamma \vdash lem_2 \quad \Gamma, \text{norm}(\varphi) \vdash \varphi_1 \subset \varphi_2}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \subset \psi(\varphi_2)} \text{congimppre}^{\subset}$$

Lemma lem_1 reads as follows: expression $f(e_1)$ can be replaced by an expression $f(e_2)$, if e_1 is equal to e_2 under the additional assumption φ . Lemma lem_2 is analogous. The proof rules make use of a function norm to normalise the additional assumption φ . The function norm turns the given formula into a set of preconditions and is defined as follows:¹

$$\begin{aligned} \text{norm}(\varphi_1 \wedge \varphi_2) &:= \text{norm}(\varphi_1) \cup \text{norm}(\varphi_2) \\ \text{norm}(\neg \varphi) &:= \{\text{neg}(\psi) \mid \psi \in \text{normneg}(\varphi)\} \\ \text{norm}(\varphi) &:= \{\varphi\} \\ \text{normneg}(\varphi_1 \vee \varphi_2) &:= \text{normneg}(\varphi_1) \cup \text{normneg}(\varphi_2) \\ \text{normneg}(\neg \varphi) &:= \{\text{neg}(\psi) \mid \psi \in \text{norm}(\varphi)\} \\ \text{normneg}(\varphi) &:= \{\varphi\} \\ \text{neg}(\neg \varphi) &:= \varphi \\ \text{neg}(\varphi) &:= \neg \varphi \end{aligned}$$

If the formula is a conjunction $\varphi_1 \wedge \varphi_2$, the two conjuncts are taken as separate preconditions. If it is a negation $\neg \varphi$, we use normneg to normalise φ and then negate the preconditions with function neg . Otherwise, formula φ is used as a single precondition $\{\varphi\}$.

Example For the operator \wedge , the following axiom is sound.

$$\text{con lem 1: } (\psi \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow (\varphi_1 \wedge \psi \rightarrow \varphi_2 \wedge \psi)$$

The conjunction $\varphi_1 \wedge \psi$ can be replaced by an equivalent or weaker formula $\varphi_2 \wedge \psi$, if subformula φ_1 implies φ_2 . The second sub-formula ψ can be used as an additional precondition while rewriting the first.

¹Read the definitions as functional program: the first equation which is applicable holds.

3.2.8 Negating congruence rules

An additional variant of a congruence rule is required to lift sub-formulas of so called negating operators.

Definition 44 (*Negating congruence rules*) *Let*

$$\text{lem: } (\varphi \rightarrow (\varphi_2 \rightarrow \varphi_1)) \rightarrow (\psi(\varphi_1) \rightarrow \psi(\varphi_2))$$

We define negating congruence rules as follows:

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma, \text{norm}(\varphi) \vdash_{\mathbf{c}} \varphi_2 \subset \varphi_1}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \psi(\varphi_1)} \text{negcongpre}^{\supset}$$

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma, \text{norm}(\varphi) \vdash_{\mathbf{c}} \varphi_1 \supset \varphi_2}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \subset \psi(\varphi_2)} \text{negcongpre}^{\subset}$$

In the second premise of $\text{negcongpre}^{\supset}$, the proof mode is swapped. If $\psi(\varphi_1)$ should imply $\psi(\varphi_2)$, then φ_2 must imply φ_1 .

Example For the operator \rightarrow , the following axiom is sound.

$$\text{imp lem 1: } (\neg \psi \rightarrow (\varphi_2 \rightarrow \varphi_1)) \rightarrow ((\varphi_1 \rightarrow \psi) \rightarrow (\varphi_2 \rightarrow \psi))$$

While descending into the first sub-formula of an implication, the proof mode is swapped. The negation of the second sub-formula ψ can be used as additional context.

3.2.9 Congruence rules with restricted context

While descending into formulas, context Γ cannot always remain unchanged.

Example Operator \exists satisfies the following congruence rule:

$$\frac{\Gamma[v_0/v] \vdash_{\mathbf{c}} \varphi_2 \supset \varphi_1}{\Gamma \vdash_{\mathbf{c}} \exists v. \varphi_2 \supset \exists v. \varphi_1} \text{ex lem}^{\supset}$$

Here, v_0 is a fresh variable which replaces v in Γ . Context Γ can be preserved; however, variable v must be renamed to avoid conflicts with the quantified variable in φ_2 .

Definition 45 (*Universal congruence rules*) *Let*

$$\text{lem: } (\forall v. \varphi_1 \rightarrow \varphi_2) \rightarrow (\psi(\varphi_1) \rightarrow \psi(\varphi_2))$$

We define universal congruence rules as follows:

$$\frac{\Gamma \vdash lem \quad \Gamma[v_0/v] \vdash_{\mathbf{c}} \varphi_2 \supset \varphi_1}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \psi(\varphi_1)} \text{allcong}^{\supset}$$

$$\frac{\Gamma \vdash lem \quad \Gamma[v_0/v] \vdash_{\mathbf{c}} \varphi_1 \subset \varphi_2}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \subset \psi(\varphi_2)} \text{allcong}^{\subset}$$

where $v_0 \notin \text{free}(\varphi_1, \varphi_2) \cup \text{free}(\Gamma) \setminus \{v\}$

For temporal operators, the context must also be adjusted. Some operators require the implication of sub-formulas to always hold ($\Box(\varphi_1 \rightarrow \varphi_2)$); others, that the implication holds on all paths ($[\text{true}] \bullet (\varphi_1 \rightarrow \varphi_2)$). Adequate rules can be found in Appendix A.

3.2.10 Weakening and strengthening rules

Definition 46 (*Weakening and strengthening rules*) Let

$$\text{lem}: \quad \varphi_1 \quad \rightarrow \quad \varphi_2$$

We define a weakening rule on the left and a strengthening rule on the right by

$$\frac{\Gamma \vdash lem}{\Gamma \vdash_{\mathbf{c}} \varphi_2 \supset \varphi_1} \text{strengthen}^{\supset} \quad \frac{\Gamma \vdash lem}{\Gamma \vdash_{\mathbf{c}} \varphi_1 \subset \varphi_2} \text{weaken}^{\subset}$$

An implication cannot be used to establish the equality of two expressions, i.e., it cannot be applied if the current proof mode is =. However, it can be used to strengthen or weaken the current formula.

Example An implication

$$\text{weaken false}: \quad \text{false} \quad \rightarrow \quad \varphi$$

can be used to effectively discard a formula. Imagine a situation

$$\vdash_{\mathbf{c}} n = 0 \vee n = 1 \supset \psi$$

where we either prove $n = 0$ or $n = 1$. If we know that we can establish $n = 1$, we can discard the first sub-formula with rule *weaken false* as follows:

$$\frac{\frac{\frac{\overline{\neg n = 1 \vdash_{\mathbf{c}} \text{false} \supset \text{false}}}{\neg n = 1 \vdash_{\mathbf{c}} n = 0 \supset \text{false}} \text{close}^{\supset}}{\vdash_{\mathbf{c}} n = 0 \vee n = 1 \supset \text{false} \vee n = 1} \text{dis lem } 1^{\supset}}{\vdash_{\mathbf{c}} n = 0 \vee n = 1 \supset \text{false} \vee n = 1} \text{weaken false}^{\supset}$$

We first use rule *dis lem 1* (which can be found in Appendix B.1.8) to descend into the first sub-formula. Then we apply *weaken false* and set the sub-formula to false. Next, rule *dis false 1* (also see Appendix B.1.8) could be applied to the resulting formula to eliminate the disjunction.

Note that lemma *weaken false* can only be used to discard formulas if applied to a positive context sequent $\Gamma \vdash_{\mathbf{c}} \varphi \supset \chi$.

3.2.11 Soundness

A complete list of rules which are used to define our rewriting approach can be found in Appendix A. The soundness of these rules is formulated in the following theorem.

Theorem 2 (*Soundness of rewriting*) *The proof rules of Appendix A are sound.*

For the proof of this theorem, a number of properties of auxiliary functions (norm, neg, etc.) are necessary. The following lemma summarises these properties.

Lemma 5 (*Soundness of normalisation of preconditions*)

1. $I \models \bigwedge \text{norm}(\varphi) \Leftrightarrow I \models \varphi$
2. $I \models \bigvee \text{normneg}(\varphi) \Leftrightarrow I \models \varphi$
3. $I \models \text{neg}(\varphi) \Leftrightarrow I \models \neg \varphi$

The properties can be proven with structural induction over formula φ . It is necessary to simultaneously prove properties 1 and 2, because functions norm and normneg are mutually recursive. Otherwise, the proof is straightforward and is therefore omitted here. \square

(Proof for Theorem 2) All rules have been verified with KIV. Proving most of the rules is straightforward. As an example, consider

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma \vdash_{\mathbf{c}} \varphi_2 \supset \varphi_1}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \psi(\varphi_1)} \text{cong}\supset$$

where

$$\text{lem:} \quad (\varphi_1 \rightarrow \varphi_2) \rightarrow (\psi(\varphi_1) \rightarrow \psi(\varphi_2)) .$$

It must be shown that the conclusion $\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \chi$ is valid under the assumption that

the premises hold. The conclusion can be transformed as follows:

$$\begin{aligned}
& \Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \psi(\varphi_1) \\
\Leftrightarrow & \models \bigwedge \Gamma \rightarrow (\psi(\varphi_1) \rightarrow \psi(\varphi_2)) && \text{Def. 38} \\
\Leftrightarrow & I \models \bigwedge \Gamma \rightarrow (\psi(\varphi_1) \rightarrow \psi(\varphi_2)) \text{ for all } I && \text{Def. 8} \\
\Leftrightarrow & (I \models \bigwedge \Gamma \text{ and } I \models \psi(\varphi_1)) \Rightarrow I \models \psi(\varphi_2) \text{ for all } I && \text{Sem. } \rightarrow
\end{aligned}$$

For an arbitrary but fixed interval I_0 , $I_0 \models \bigwedge \Gamma$ and $I_0 \models \psi(\varphi_1)$ can be assumed. It remains to be shown that $I_0 \models \psi(\varphi_2)$. We assume that $I_0 \not\models \psi(\varphi_2)$. With the same definitions as above, the premises can be transformed, too. Following from premise 1

$$(I \models \varphi_1 \Rightarrow I \models \varphi_2) \text{ and } I \models \psi(\varphi_1) \Rightarrow I \models \psi(\varphi_2) \text{ for all } I$$

For interval I_0 , because $I_0 \not\models \psi(\varphi_2)$, either $I_0 \not\models \psi(\varphi_1)$ or not ($I_0 \models \varphi_1 \Rightarrow I_0 \models \varphi_2$). The second alternative contradicts premise 2: the premise reads

$$I \models \bigwedge \Gamma \text{ and } I \models \varphi_1 \Rightarrow I \models \varphi_2 \text{ for all } I .$$

For interval I_0 , $I_0 \models \bigwedge \Gamma$ and therefore $I_0 \models \varphi_1 \Rightarrow I_0 \models \varphi_2$. Thus, the first alternative $I_0 \not\models \psi(\varphi_1)$ holds, which contradicts one of our initial assumptions.

Proofs for congruence rules with additional preconditions are very similar; however, the proof involves properties of Lemma 5. \square

Besides the basic rules of Appendix A, a number of rules can be derived to simplify proofs.

Lemma 6 (Derived proof rules) *For any proof mode $* \in \{\supset, \subset, =\}$, the following rules can be derived:*

$$\frac{}{\Gamma_1, \varphi, \Gamma_2 \vdash_{\mathbf{c}} \varphi * \text{true}} \text{ ax true}^* \quad \frac{}{\Gamma_1, \neg \varphi, \Gamma_2 \vdash_{\mathbf{c}} \varphi * \text{true}} \text{ ax false}^*$$

As an example, we present a derivation of rule ax true^{\supset} . Derivations of other rules are very similar. Let $\Gamma \equiv \Gamma_1, \varphi, \Gamma_2$. Starting with the conclusion $\Gamma \vdash_{\mathbf{c}} \varphi \supset \text{true}$, we can derive

$$\frac{\frac{\frac{}{\Gamma \vdash_{\mathbf{c}} \varphi \leftrightarrow \text{true} \supset \varphi} \text{ eqv true } 2^{\supset} \quad \frac{}{\Gamma \vdash_{\mathbf{c}} \varphi} \text{ ax}}{\Gamma \vdash_{\mathbf{c}} \varphi \leftrightarrow \text{true}} \text{ rewrite}}{\Gamma \vdash_{\mathbf{c}} \varphi \supset \text{true}} \text{ rw}^{\supset}$$

\square

3.2.12 Context sensitive rule application

Rewriting sub-formulas as described above requires a number of rules: first we apply *rewrite* to initiate rewriting, second we apply a number of congruence rules to lift the desired sub-formula to top level, third the rule to rewrite the sub formula is applied.

Example Consider the proof obligation $A \rightarrow A$. In order to rewrite the right hand side of the implication to true with rule *ax true*, we need to apply

$$\frac{\frac{\overline{A \vdash_{\mathbf{c}} A \supset \text{true}} \quad 3^{\supset}}{\vdash_{\mathbf{c}} A \rightarrow A \supset A \rightarrow \text{true}} \quad 2^{\supset} \quad \vdash A \rightarrow \text{true}}{\vdash A \rightarrow A} \quad 1$$

where $1 \equiv \text{rewrite}$, $2 \equiv \text{imp lem}$, and $3 \equiv \text{ax true}$.

We shall abbreviate the proof tree as follows: the sub formula which is to be rewritten is shaded, and the congruence rules which are necessary to lift the sub formula are omitted. Two rules remain: the rule *rewrite* to initiate rewriting and the actual rewrite rule for the sub formula. The following example illustrates our notation.

Example The proof tree of the example above is abbreviated by

$$\frac{\overline{A \vdash_{\mathbf{c}} A \supset \text{true}} \quad \text{ax true}^{\supset} \quad \vdash A \rightarrow \text{true}}{\vdash A \rightarrow A} \quad \text{rewrite}$$

In the first premise, the formula to be rewritten is the top level formula. The context is the local context which can be used to rewrite the formula, and the resulting formula is the result of rewriting

If the local context is trivial or dispensable, we often use an even shorter notation by omitting the first premise and using the name of the rewrite rule as rule name.

Example The example in an even shorter notation reads:

$$\frac{\vdash A \rightarrow \text{true}}{\vdash A \rightarrow A} \quad \text{ax true}^{\supset}$$

This notation is unambiguous, if there is at most one congruence rule for each parameter of each operator and for each proof mode. However, our calculus defines properties

$$\begin{aligned} \text{op rw:} \quad & \varphi_1 \leftrightarrow \varphi_2 \rightarrow (\text{op}(\varphi_1) \leftrightarrow \text{op}(\varphi_2)) \\ \text{op lem:} \quad & \varphi_1 \rightarrow \varphi_2 \rightarrow (\text{op}(\varphi_1) \rightarrow \text{op}(\varphi_2)) \end{aligned}$$

for a number of operators *op*. These properties are useful to apply both rules *cong*[⊃] (resp. *cong*[⊂]) and *congimp*[⊃] (resp. *congimp*[⊂]). In this case, the latter rule *congimp*[⊃] (resp. *congimp*[⊂]) is preferred.

We refer to the abbreviated notation as *context sensitive rule application*: a sub formula is selected and an appropriate rewrite rule is applied.

$$\begin{array}{c}
\frac{}{\varphi, \Gamma \vdash_{\mathbf{G}} \varphi, \Delta} \text{ ax} \\
\\
\frac{}{\text{false}, \Gamma \vdash_{\mathbf{G}} \Delta} \text{ false l} \quad \frac{}{\Gamma \vdash_{\mathbf{G}} \text{true}, \Delta} \text{ true r} \\
\\
\frac{\Gamma, \varphi \vdash_{\mathbf{G}} \Delta}{\varphi, \Gamma \vdash_{\mathbf{G}} \Delta} \text{ rot l} \quad \frac{\Gamma \vdash_{\mathbf{G}} \Delta, \varphi}{\Gamma \vdash_{\mathbf{G}} \varphi, \Delta} \text{ rot r} \\
\\
\frac{\Gamma \vdash_{\mathbf{G}} \varphi, \Delta}{\neg \varphi, \Gamma \vdash_{\mathbf{G}} \Delta} \text{ not l} \quad \frac{\varphi, \Gamma \vdash_{\mathbf{G}} \Delta}{\Gamma \vdash_{\mathbf{G}} \neg \varphi, \Delta} \text{ not r} \\
\\
\frac{\varphi_1, \varphi_2, \Gamma \vdash_{\mathbf{G}} \Delta}{\varphi_1 \wedge \varphi_2, \Gamma \vdash_{\mathbf{G}} \Delta} \text{ con l} \quad \frac{\Gamma \vdash_{\mathbf{G}} \varphi_1, \Delta \quad \Gamma \vdash_{\mathbf{G}} \varphi_2, \Delta}{\Gamma \vdash_{\mathbf{G}} \varphi_1 \wedge \varphi_2, \Delta} \text{ con r} \\
\\
\frac{\varphi_1, \Gamma \vdash_{\mathbf{G}} \Delta \quad \varphi_2, \Gamma \vdash_{\mathbf{G}} \Delta}{\varphi_1 \vee \varphi_2, \Gamma \vdash_{\mathbf{G}} \Delta} \text{ dis l} \quad \frac{\Gamma \vdash_{\mathbf{G}} \varphi_1, \varphi_2, \Delta}{\Gamma \vdash_{\mathbf{G}} \varphi_1 \vee \varphi_2, \Delta} \text{ dis r} \\
\\
\frac{\Gamma \vdash_{\mathbf{G}} \varphi_1, \Delta \quad \varphi_2, \Gamma \vdash_{\mathbf{G}} \Delta}{\varphi_1 \rightarrow \varphi_2, \Gamma \vdash_{\mathbf{G}} \Delta} \text{ imp l} \quad \frac{\varphi_1, \Gamma \vdash_{\mathbf{G}} \varphi_2, \Delta}{\Gamma \vdash_{\mathbf{G}} \varphi_1 \rightarrow \varphi_2, \Delta} \text{ imp r} \\
\\
\frac{\varphi_1, \varphi_2, \Gamma \vdash_{\mathbf{G}} \Delta \quad \Gamma \vdash_{\mathbf{G}} \varphi_1, \varphi_2, \Delta}{\varphi_1 \leftrightarrow \varphi_2, \Gamma \vdash_{\mathbf{G}} \Delta} \text{ equiv l} \\
\\
\frac{\varphi_1, \Gamma \vdash_{\mathbf{G}} \varphi_2, \Delta \quad \varphi_2, \Gamma \vdash_{\mathbf{G}} \varphi_1, \Delta}{\Gamma \vdash_{\mathbf{G}} \varphi_1 \leftrightarrow \varphi_2, \Delta} \text{ equiv r}
\end{array}$$

Table 3.19: Sequent calculus for propositional logic

3.3 Example: Propositional Logic

Table 3.19 displays a set of rules of a standard sequent calculus for propositional logic.² The goal of this section is to define rewrite rules which can be used to “emulate” these rules. We shall not try to come up with a minimal set of rules, and will focus on automatic application of rules instead.

3.3.1 Rewrite rules

The first set of properties in Table 3.21 defines congruence rules. They can be used to lift sub formulas to top level. For every operator except \leftrightarrow , there are two properties for every sub formula $\langle op \rangle rw i$ and $\langle op \rangle lem i$, the first being applied in mode $=$, the second defining rules for \supset and \subset . For operator \leftrightarrow , sub formulas can only be replaced by equivalent sub-formulas. The congruence rules are not manually applied, but are implicitly used, if a sub formula is to be rewritten (compare Section 3.2.12).

²We shall use symbol $\vdash_{\mathbf{G}}$ to distinguish between a sequent of a Gentzen calculus and a sequent as is defined for our purposes.

<i>not rw:</i>	$(\varphi_1 \leftrightarrow \varphi_2) \rightarrow (\neg \varphi_1 \leftrightarrow \neg \varphi_2)$
<i>not lem:</i>	$(\varphi_2 \rightarrow \varphi_1) \rightarrow (\neg \varphi_1 \rightarrow \neg \varphi_2)$
<i>con rw 1:</i>	$(\psi \rightarrow (\varphi_1 \leftrightarrow \varphi_2)) \rightarrow (\varphi_1 \wedge \psi \leftrightarrow \varphi_2 \wedge \psi)$
<i>con lem 1:</i>	$(\psi \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow (\varphi_1 \wedge \psi \rightarrow \varphi_2 \wedge \psi)$
<i>con rw 2:</i>	$(\psi \rightarrow (\varphi_1 \leftrightarrow \varphi_2)) \rightarrow (\psi \wedge \varphi_1 \leftrightarrow \psi \wedge \varphi_2)$
<i>con lem 2:</i>	$(\psi \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow (\psi \wedge \varphi_1 \rightarrow \psi \wedge \varphi_2)$
<i>dis rw 1:</i>	$(\neg \psi \rightarrow (\varphi_1 \leftrightarrow \varphi_2)) \rightarrow (\varphi_1 \vee \psi \leftrightarrow \varphi_2 \vee \psi)$
<i>dis lem 1:</i>	$(\neg \psi \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow (\varphi_1 \vee \psi \rightarrow \varphi_2 \vee \psi)$
<i>dis rw 2:</i>	$(\neg \psi \rightarrow (\varphi_1 \leftrightarrow \varphi_2)) \rightarrow (\psi \vee \varphi_1 \leftrightarrow \psi \vee \varphi_2)$
<i>dis lem 2:</i>	$(\neg \psi \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow (\psi \vee \varphi_1 \rightarrow \psi \vee \varphi_2)$
<i>imp rw 1:</i>	$(\neg \psi \rightarrow (\varphi_1 \leftrightarrow \varphi_2)) \rightarrow ((\varphi_1 \rightarrow \psi) \leftrightarrow (\varphi_2 \rightarrow \psi))$
<i>imp lem 1:</i>	$(\neg \psi \rightarrow (\varphi_2 \rightarrow \varphi_1)) \rightarrow ((\varphi_1 \rightarrow \psi) \rightarrow (\varphi_2 \rightarrow \psi))$
<i>imp rw 2:</i>	$(\psi \rightarrow (\varphi_1 \leftrightarrow \varphi_2)) \rightarrow ((\psi \rightarrow \varphi_1) \leftrightarrow (\psi \rightarrow \varphi_2))$
<i>imp lem 2:</i>	$(\psi \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow ((\psi \rightarrow \varphi_1) \rightarrow (\psi \rightarrow \varphi_2))$
<i>eqv rw 1:</i>	$(\varphi_1 \leftrightarrow \varphi_2) \rightarrow ((\varphi_1 \leftrightarrow \psi) \leftrightarrow (\varphi_2 \leftrightarrow \psi))$
<i>eqv rw 2:</i>	$(\varphi_1 \leftrightarrow \varphi_2) \rightarrow ((\psi \leftrightarrow \varphi_1) \leftrightarrow (\psi \leftrightarrow \varphi_2))$

Table 3.21: Congruence rules

<i>not true:</i>	$\neg \text{true} \leftrightarrow \text{false}$
<i>not false:</i>	$\neg \text{false} \leftrightarrow \text{true}$
<i>con true 1:</i>	$\text{true} \wedge \varphi \leftrightarrow \varphi$
<i>con false 1:</i>	$\text{false} \wedge \varphi \leftrightarrow \text{false}$
<i>con true 2:</i>	$\varphi \wedge \text{true} \leftrightarrow \varphi$
<i>con false 2:</i>	$\varphi \wedge \text{false} \leftrightarrow \text{false}$
<i>dis true 1:</i>	$\text{true} \vee \varphi \leftrightarrow \text{true}$
<i>dis false 1:</i>	$\text{false} \vee \varphi \leftrightarrow \varphi$
<i>dis true 2:</i>	$\varphi \vee \text{true} \leftrightarrow \text{true}$
<i>dis false 2:</i>	$\varphi \vee \text{false} \leftrightarrow \varphi$
<i>imp true 1:</i>	$\text{true} \rightarrow \varphi \leftrightarrow \varphi$
<i>imp false 1:</i>	$\text{false} \rightarrow \varphi \leftrightarrow \text{true}$
<i>imp true 2:</i>	$\varphi \rightarrow \text{true} \leftrightarrow \text{true}$
<i>imp false 2:</i>	$\varphi \rightarrow \text{false} \leftrightarrow \neg \varphi$
<i>eqv true 1:</i>	$(\text{true} \leftrightarrow \varphi) \leftrightarrow \varphi$
<i>eqv false 1:</i>	$(\text{false} \leftrightarrow \varphi) \leftrightarrow \neg \varphi$
<i>eqv true 2:</i>	$(\varphi \leftrightarrow \text{true}) \leftrightarrow \varphi$
<i>eqv false 2:</i>	$(\varphi \leftrightarrow \text{false}) \leftrightarrow \neg \varphi$

Table 3.23: Simplification of true and false

<i>not not:</i>	$\neg \neg \varphi$	\leftrightarrow	φ
<i>not connot:</i>	$\neg (\varphi_1 \wedge \neg \varphi_2)$	\leftrightarrow	$\varphi_1 \rightarrow \varphi_2$
<i>con not 1:</i>	$\neg \varphi \wedge \psi^+$	\leftrightarrow	$\psi^+ \wedge \neg \varphi$
<i>con connot 2:</i>	$\psi \wedge (\varphi_1 \wedge \neg \varphi_2)$	\leftrightarrow	$(\psi \wedge \varphi_1) \wedge \neg \varphi_2$
<i>con connot 1:</i>	$(\varphi_1 \wedge \neg \varphi_2) \wedge \psi^+$	\leftrightarrow	$(\varphi_1 \wedge \psi^+) \wedge \neg \varphi_2$
<i>dis not 1:</i>	$\neg \varphi \vee \psi$	\leftrightarrow	$\varphi \rightarrow \psi$
<i>dis not 2:</i>	$\psi \vee \neg \varphi$	\leftrightarrow	$\varphi \rightarrow \psi$
<i>dis imp 1:</i>	$(\varphi_1 \rightarrow \varphi_2) \vee \psi$	\leftrightarrow	$\varphi_1 \rightarrow \varphi_2 \vee \psi$
<i>dis imp 2:</i>	$\psi \vee (\varphi_1 \rightarrow \varphi_2)$	\leftrightarrow	$\varphi_1 \rightarrow \psi \vee \varphi_2$
<i>imp not 1:</i>	$\neg \varphi \rightarrow \psi$	\leftrightarrow	$\varphi \vee \psi$
<i>imp not 2:</i>	$\psi \rightarrow \neg \varphi$	\leftrightarrow	$\neg (\psi \wedge \varphi)$
<i>imp connot 1:</i>	$\varphi_1 \wedge \neg \varphi_2 \rightarrow \psi$	\leftrightarrow	$\varphi_1 \rightarrow \varphi_2 \vee \psi$
<i>imp imp 2:</i>	$\psi \rightarrow \varphi_1 \rightarrow \varphi_2$	\leftrightarrow	$\psi \wedge \varphi_1 \rightarrow \varphi_2$

where ψ^+ is a positive formula, i.e., a formula without leading \neg operator.

Table 3.25: Simplification of combinations of operators

The second set of rules in Table 3.23 defines how to simplify operators if one of the parameters is true or false. These rules are automatically applied. They can be used to “emulate” sequent rules *false l* and *true r*.

The third set of rules in Table 3.25 are used to simplify combinations of operators. The simplification strategy is close to the strategy of a sequent calculus: rearrange formulas such that we receive an implication with a conjunction of (positive) formulas on the left and a disjunction of (positive) formulas on the right. These rules are automatically applied and can be used to “emulate” sequent rules *not l*, *not r*, *imp l*, and *imp r*. Some rules require a formula ψ^+ to be positive. This is to ensure that automatic application of rules terminates.

The fourth set of rules in Table 3.27 can be used to split cases. These rules are applied as follows: an operator which gives a case distinction is manually selected and the rewrite rules are automatically applied to lift the different cases to receive a conjunction of formulas on top level. Rule *con* generates premises for each of the top level conjuncts.

3.3.2 Example

Example Consider formula

$$(A \rightarrow B) \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B) \quad (3.1)$$

which corresponds to a sequent $A \rightarrow B, B \rightarrow A \vdash_{\mathcal{G}} A \leftrightarrow B$. For this sequent, the sequent

<i>not dis:</i>	$\neg (\varphi_1 \vee \varphi_2)$	\leftrightarrow	$\neg \varphi_1 \wedge \neg \varphi_2$
<i>con dis 1:</i>	$(\varphi_1 \vee \varphi_2) \wedge \psi$	\leftrightarrow	$\varphi_1 \wedge \psi \vee \varphi_2 \wedge \psi$
<i>con dis 2:</i>	$\psi \wedge (\varphi_1 \vee \varphi_2)$	\leftrightarrow	$\psi \wedge \varphi_1 \vee \psi \wedge \varphi_2$
<i>dis con 1:</i>	$\varphi_1 \wedge \varphi_2 \vee \psi$	\leftrightarrow	$(\varphi_1 \vee \psi) \wedge (\varphi_2 \vee \psi)$
<i>dis con 2:</i>	$\psi \vee \varphi_1 \wedge \varphi_2$	\leftrightarrow	$(\psi \vee \varphi_1) \wedge (\psi \vee \varphi_2)$
<i>imp dis 1:</i>	$\varphi_1 \vee \varphi_2 \rightarrow \psi$	\leftrightarrow	$(\varphi_1 \rightarrow \psi) \wedge (\varphi_2 \rightarrow \psi)$
<i>imp con 2:</i>	$\psi \rightarrow \varphi_1 \wedge \varphi_2$	\leftrightarrow	$(\psi \rightarrow \varphi_1) \wedge (\psi \rightarrow \varphi_2)$
<i>imp case:</i>	$\varphi_1 \rightarrow \varphi_2$	\leftrightarrow	$\neg \varphi_1 \vee \varphi_2$
<i>eqv case:</i>	$(\varphi_1 \leftrightarrow \varphi_2)$	\leftrightarrow	$\varphi_1 \wedge \varphi_2 \vee \neg \varphi_1 \wedge \neg \varphi_2$
<i>eqv case:</i>	$(\varphi_1 \leftrightarrow \varphi_2)$	\leftrightarrow	$(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$

Table 3.27: Case distinction rules

rule *imp l* can be used to receive two premises

$$\frac{B \rightarrow A \vdash_{\mathbf{G}} A, A \leftrightarrow B \quad B, B \rightarrow A \vdash_{\mathbf{G}} A \leftrightarrow B}{A \rightarrow B, B \rightarrow A \vdash_{\mathbf{G}} A \leftrightarrow B} \text{imp } l$$

In order to emulate this rule with rewrite rules, we apply *imp case* to the sub implication of (3.1) to receive

$$\frac{\vdash (\neg A \vee B) \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)}{\vdash (A \rightarrow B) \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)} \text{imp case}$$

Afterwards, the disjunction is lifted with a successive application of the two rules *con dis 1* and *imp dis 1*.

$$\frac{\vdash (\neg A \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)) \wedge (B \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B))}{\vdash \neg A \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B) \vee B \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)} \text{imp dis 1}$$

$$\frac{\vdash \neg A \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B) \vee B \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)}{\vdash (\neg A \vee B) \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)} \text{con dis 1}$$

⋮

This gives a conjunction on top level and rule *con* can be used to generate two premises.

$$\frac{\vdash \neg A \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B) \quad \vdash B \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)}{\vdash (\neg A \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)) \wedge (B \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B))} \text{con}$$

⋮

The resulting premises correspond closely to the premises of rule *imp l* with the single difference that in the first premise, formula $\neg A$ is still on the left hand side of the implication.

For a shorter notation, after a rule of Table 3.27 has been applied, appropriate rules to lift the resulting cases to top level and the final rule *con* to generate premises are implicitly applied.

Example In short, we write

$$\frac{\vdash \neg A \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B) \quad \vdash B \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)}{\vdash (A \rightarrow B) \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)} \text{ imp case}$$

Rules to lift the resulting disjunction of rule *imp case* the top level and to generate premises for the two cases are implicitly applied.

The example is continued with shifting the negated formula $\neg A$ of the first premise to the right hand side of the implication as follows:

$$\frac{\frac{\frac{\vdash (B \rightarrow A) \rightarrow A \vee (A \leftrightarrow B)}{\vdash (B \rightarrow A) \wedge \neg A \rightarrow (A \leftrightarrow B)} \text{ imp connot 1}}{\vdash \neg A \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)} \text{ con not 1}}{\vdash \neg A \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)} \text{ :}$$

The resulting premise finally corresponds to the premise of rule *imp l*.

The rules which were used to shift the negated formula can be automatically applied as these rules strictly simplify a given formula. Therefore, we often abstract the rule applications with a single rule *simp*.

Example In our example, we may write

$$\frac{\frac{\vdash (B \rightarrow A) \rightarrow A \vee (A \leftrightarrow B)}{\vdash \neg A \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)} \text{ simp}}{\vdash \neg A \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)} \text{ :}$$

The premise of the example can be further simplified. The congruence rules of Table 3.21 make it possible to use $\neg A$ as additional context while rewriting the other sub formulas. Thus, the marked occurrence of A can be rewritten to false with rule *ax false* of Lemma 6.

$$\frac{\frac{\overline{\neg A, \dots \vdash_c A \subset \text{false}} \text{ ax false}^c \quad \vdash (B \rightarrow \text{false}) \rightarrow A \vee (A \leftrightarrow B)}{\vdash (B \rightarrow A) \rightarrow A \vee (A \leftrightarrow B)} \text{ rewrite}}$$

The other occurrence of A can also be rewritten:

$$\frac{\overline{\dots, \neg A \vdash_c A = \text{false}} \text{ ax false}^= \quad \vdash (B \rightarrow \text{false}) \rightarrow A \vee (\text{false} \leftrightarrow B)}{\vdash (B \rightarrow \text{false}) \rightarrow A \vee (A \leftrightarrow B)} \text{ rewrite}$$

With rules from Table 3.23, we receive

$$\frac{\vdash \neg B \rightarrow A \vee \neg B}{\vdash (B \rightarrow \text{false}) \rightarrow A \vee (\text{false} \leftrightarrow B)} \text{ simp}$$

Rules from Table 3.25 can be used to shift the negated formulas. In detail, this is as follows:

$$\frac{\frac{\frac{\vdash B \rightarrow B \vee A}{\vdash B \vee (B \rightarrow A)} \text{dis imp } 2}{\vdash B \vee (A \vee \neg B)} \text{dis not } 2}{\vdash \neg B \rightarrow A \vee \neg B} \text{imp not } 1$$

It is now possible to replace the occurrence of B on the right hand side with true

$$\frac{\frac{\overline{B, \neg A \vdash_c B \supset \text{true}} \text{ax true}^\supset \quad \vdash B \rightarrow \text{true} \vee A}{\vdash B \rightarrow B \vee A} \text{rewrite}}$$

which again can be simplified to receive

$$\frac{\overline{\vdash \text{true}} \text{true}}{\vdash B \rightarrow \text{true} \vee A} \text{simp}$$

The final premise can be closed with rule true .

If it is not too difficult to understand, we shall represent multiple applications of rules of Tables 3.23 and 3.25 as well as rules ax true^* , ax false^* and true with a single rule simp .

Example Formula (3.1) can be proven as follows:

$$\frac{\frac{\overline{\vdash \neg A \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)} \text{simp} \quad \overline{\vdash B \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)} \text{simp}}{\vdash (A \rightarrow B) \wedge (B \rightarrow A) \rightarrow (A \leftrightarrow B)} \text{imp case}}$$

3.3.3 Soundness and completeness

Lemma 7 (*Soundness of properties for Propositional Logic*) *The properties of Tables 3.21, 3.23, 3.25, and 3.27 are sound.*

All properties have been verified with KIV. It is straightforward but lengthy to prove all of the different properties. Proves are skipped here. \square

Lemma 8 (*Completeness of properties for Propositional Logic*) *Given the properties of Tables 3.21, 3.23, 3.25, and 3.27 together with rules of Appendix A. For any propositional formula φ*

$$\models \varphi \Rightarrow \vdash \varphi$$

The sequent calculus of Table 3.19 is complete, thus it holds

$$\models \varphi \Rightarrow \vdash_c \varphi$$

Rule *equiv l*

$$\begin{array}{c}
 \vdots \\
 \hline
 \frac{\vdash \dots \quad \vdash \bigwedge \Gamma_1 \wedge \neg \varphi_1 \wedge \neg \varphi_2 \wedge \bigwedge \Gamma_2 \rightarrow \bigvee \Delta}{\vdash (\dots) \wedge (\bigwedge \Gamma_1 \wedge \neg \varphi_1 \wedge \neg \varphi_2 \wedge \bigwedge \Gamma_2 \rightarrow \bigvee \Delta)} \text{con} \\
 \hline
 \frac{\vdash \dots \quad \bigvee \bigwedge \Gamma_1 \wedge \neg \varphi_1 \wedge \neg \varphi_2 \wedge \bigwedge \Gamma_2 \rightarrow \bigvee \Delta}{\vdash \bigwedge \Gamma_1 \wedge (\dots \vee \neg \varphi_1 \wedge \neg \varphi_2 \wedge \bigwedge \Gamma_2) \rightarrow \bigvee \Delta} \text{imp dis 1} \\
 \hline
 \frac{\vdash \bigwedge \Gamma_1 \wedge (\dots \vee \neg \varphi_1 \wedge \neg \varphi_2 \wedge \bigwedge \Gamma_2) \rightarrow \bigvee \Delta}{\vdash \bigwedge \Gamma_1 \wedge (\varphi_1 \wedge \varphi_2 \vee \neg \varphi_1 \wedge \neg \varphi_2) \wedge \bigwedge \Gamma_2 \rightarrow \bigvee \Delta} \text{con dis 2} \\
 \hline
 \frac{\vdash \bigwedge \Gamma_1 \wedge (\varphi_1 \wedge \varphi_2 \vee \neg \varphi_1 \wedge \neg \varphi_2) \wedge \bigwedge \Gamma_2 \rightarrow \bigvee \Delta}{\vdash \bigwedge \Gamma_1 \wedge (\varphi_1 \leftrightarrow \varphi_2) \wedge \bigwedge \Gamma_2 \rightarrow \bigvee \Delta} \text{equiv case}
 \end{array}$$

□

3.4 Conclusion

Rewriting can be used to apply rules to sub formulas. Instead of simply replacing all “free occurrences” of formulas with equivalent formulas, we have defined a more elaborate rewrite method: congruence rules describe how to preserve context while descending into sub formulas. The context basically is a list of preconditions which can be used to establish the conditions of conditional rewrite rules. The use of different proof modes (see Def. 37) makes it possible to not only replace sub formulas with equivalent formulas but also to weaken or strengthen parts. This is important for abstraction in Chapter 7.

The individual definitions of the chapter show how to derive rules from properties. This turned out to be a good approach for two reasons: (1) several rules (with different proof mode) can be derived from a single property, (2) the soundness of rules can be established by verifying the property from which the rule has been derived. The application of rules is a bit strange as the resulting expression or formula in a context sequent is not immediately instantiated and meta variables are used instead. An alternative would be to define rewriting of formulas in an algorithmic style, e.g.,

$$\text{apply}(\Gamma, *, \varphi_1, \text{path}, \text{rule}) = \varphi_2$$

where Γ is the set of preconditions, $*$ is the current proof mode, φ_1 is the formula to be rewritten, *path* somehow defines the path to the sub formula of φ_1 which should be rewritten with a given *rule*; the resulting formula is φ_2 . The use of congruence rules, however, is probably more flexible to describe how rewriting descends into the formula and how the context information is adjusted.

As an example, we have defined a set of rewrite rules to reason in Propositional Logic. The rules emulate the strategy of a sequent calculus. However, they can also be used to effectively simplify the whole formula: congruence rules collect additional context which can be exploited with rules *ax true* and *ax false* while simplifying sub formulas. Most of the rules can be automatically applied as they do not enlarge the given formula. Rules leading to case distinctions – and therefore to a larger formula – can be manually applied. The given set of rules is sound and complete.

Chapter 4

Ingredient 1 – Symbolic Execution

In this chapter, rewrite rules are defined to symbolically execute temporal formulas, which includes the execution of parallel programs. Normal forms are introduced in Sect. 4.2. The sections following this section explain how to execute the different operators with focus on the execution of interleaving.

4.1 Idea

Our basic idea of symbolic execution is to rewrite a program such that part of the resulting formula describes the transition to execute next and the other part represents the program configuration to continue with in the next step. More formally, a program (or temporal formula) φ should be rewritten to a formula of the following type

$$\tau \wedge \circ \psi$$

with τ describing a transition as a PL relation between unprimed, primed and doubly primed variables. In general, a program may also terminate, i.e., under certain conditions, the current state may be the last. Furthermore, the next transition can be nondeterministic, i.e., different τ_i with corresponding ψ_i may exist describing the possible transitions and corresponding next steps. Finally, there may exist a link between the transition τ_i and system ψ_i which cannot be expressed as a relation between unprimed, primed, and doubly primed variables in the transition alone. This link is captured in existentially quantified static variables \vec{X}_i which occur in both τ_i and ψ_i . The general pattern resulting from rewriting a formula φ is

$$\tau_0 \wedge \mathbf{last} \vee \left(\bigvee_{i=1}^n (\exists \vec{X}_i. \tau_i \wedge \circ \psi_i) \right) .$$

We will refer to this general pattern as normal form. Rewriting a formula to an equivalent formula in normal form is what we call symbolic execution.

In our setting, there is more to a transition τ than just relations in predicate logic. Consider an assignment $x := e$ which is by definition equivalent to the following formula (see Sect. 2.2.7)

$$x' = e \wedge [x] \wedge \circ \mathbf{last}$$

The frame assumption $[x]$ can be interpreted as a formula in predicate logic; however, the formula would be infinitely large (see Sect. 1.2.2) and must therefore be treated special. For the execution of interleaving, it is also of special interest whether a transition is blocked. This is why frame assumptions and properties for variable \mathbf{blk} are separated in a transition as is defined in the next section.

4.2 Normal form

A transition τ is a formula which adheres to a special format. The transition consists of a transition predicate ρ , a frame formula δ , and a block statement β . These three parts are either combined as conjunction or disjunction to form the overall transition. More formally, τ is defined to be as follows.

Definition 47 (Transition) Let $v \in \mathbf{X} \cup \mathbf{Y}$, $w \in \mathbf{Y} \cup \mathbf{Z}$.

1. A transition predicate ρ is as follows:

$$\rho ::= X \mid w \mid w' \mid w'' \mid f(\rho, \dots, \rho) \mid \rho = \rho \mid \exists v. \rho .$$

where $y \neq \mathbf{blk}$.

2. A frame formula δ is as follows:

$$\delta ::= \mathbf{true} \mid \mathbf{false} \mid [x_1, \dots, x_n] \mid \neg [x_1, \dots, x_n]$$

3. A block statement β is as follows:

$$\beta ::= \mathbf{true} \mid \mathbf{false} \mid \mathbf{blocked} \mid \neg \mathbf{blocked}$$

4. A transition τ is as follows:

$$\tau^c ::= \rho \wedge \delta \wedge \beta$$

$$\tau^d ::= \rho \vee \delta \vee \beta$$

$$\tau ::= \tau^c \mid \tau^d .$$

Formulas τ^c (resp. τ^d) are called conjunctive (resp. disjunctive) transitions.

The two notions of conjunctive and disjunctive transitions are necessary to define disjunctive and conjunctive normal forms. The different normal forms in turn are necessary to execute the different types of operators as is explained in the following sections.

Definition 48 (*Normal form*)

1. A formula is in disjunctive normal form if and only if it adheres to the following pattern

$$\rho_0 \wedge \mathbf{last} \vee \left(\bigvee_{i=1}^n (\exists \vec{X}_i. \tau_i^c \wedge \circ \psi_i) \right) .$$

2. A formula is in conjunctive normal form if and only if it adheres to the following pattern

$$(\rho_0 \vee \neg \mathbf{last}) \wedge \left(\bigwedge_{i=1}^n (\forall \vec{X}_i. \tau_i^d \vee \bullet \psi_i) \right) .$$

If a formula is in disjunctive normal form, the behavior of the system in the next step is explicit. The system either terminates or takes one of a number of transitions. The system can only terminate, if condition ρ_0 holds. The different transition relations are described in formulas τ_i^c . For each transition, there is a formula ψ_i which represents the system configuration in the next state. The dual formula to a disjunctive normal form is a conjunctive normal form.

It is an important property of a transition τ that only the first states $I(0)$, $I(0)'$, and $I(0)''$ of an interval are relevant for its interpretation. This is expressed in the following lemma.

Lemma 9 (*Transition*) A transition τ and a transition predicate ρ satisfy

1. if $|I| > 0$, then $I \models \tau \Leftrightarrow (I(0), I(0)', I(0)'') \models \tau$
2. if $|I| = 0$, then $I \models \tau \Leftrightarrow (I(0), I(0), I(0)) \models \tau$
3. if $|I| = 0$, then $I \models \rho \Leftrightarrow I \models \rho^{[w,w/w',w'']}$

Structural induction for transitions τ and transition predicates ρ (see Def. 47 for the structure of transitions and transition predicates). \square

Rewriting is used to transform a formula to normal form. In the following sections, appropriate rewrite rules are defined for the different operators. Here, we give a number of basic rewrite rules which can be used to rearrange the formulas of a disjunction (resp. conjunction) to receive a formula which strictly adheres to normal form.

Lemma 10 (*Rearranging normal forms*) The following rules which are used to rearrange the formulas in a disjunction (resp. conjunction), to arrive with a formula which strictly adheres to normal form, are sound.

$$\begin{array}{llll}
dnf\ swp: & \tau \wedge \circ \varphi \vee \rho \wedge \mathbf{last} & \leftrightarrow & \rho \wedge \mathbf{last} \vee \tau \wedge \circ \varphi \\
dnf\ lsts: & \rho_1 \wedge \mathbf{last} \vee \rho_2 \wedge \mathbf{last} & \leftrightarrow & (\rho_1 \vee \rho_2) \wedge \mathbf{last} \\
cnf\ swp: & (\tau \vee \bullet \varphi) \wedge (\rho \vee \neg \mathbf{last}) & \leftrightarrow & (\rho \vee \neg \mathbf{last}) \wedge (\tau \vee \circ \varphi) \\
cnf\ lsts: & (\rho_1 \vee \neg \mathbf{last}) \wedge (\rho_2 \vee \neg \mathbf{last}) & \leftrightarrow & \rho_1 \wedge \rho_2 \vee \neg \mathbf{last}
\end{array}$$

Rules *dnf swp* and *cnf swp* simply rely on commutativity of disjunction and conjunction. Rules *dnf lsts* and *cnf lsts* are sound, because conjunction distributes over disjunction and vice versa. \square

4.2.1 Weak normal form

The different cases of a disjunctive normal form either terminate ($\rho_0 \vee \mathbf{last}$) or describe a transition ($\tau_i^c \wedge \circ \psi_i$). This definition can be relaxed to allow for cases in which it is undefined whether the system terminates or steps ($\tau_i^c \wedge \bullet \psi_i$).

Definition 49 (*Weak normal form*)

1. A formula is in weak disjunctive normal form if and only if it adheres to the following pattern

$$\rho_0 \wedge \mathbf{last} \vee \left(\bigvee_{i=1}^n (\exists \vec{X}_i. \tau_i^c \wedge (\circ \psi_i \mid \bullet \psi_i)) \right) .$$

2. A formula is in weak conjunctive normal form if and only if it adheres to the following pattern

$$(\rho_0 \vee \neg \mathbf{last}) \wedge \left(\bigwedge_{i=1}^n (\forall \vec{X}_i. \tau_i^d \vee (\bullet \psi_i \mid \circ \psi_i)) \right) .$$

A weak normal form is more general and often smaller than the comparable “strong” normal form. To rewrite a weak normal form to normal form, the weak next (resp. strong next) operator must be eliminated. This is achieved by the following rules which introduce different cases for the weak next (resp. strong next) operator.

Lemma 11 (*Conversion of weak normal form*) *The following rules, which are used to convert weak normal forms to normal forms, are sound.*

$$\begin{array}{llll}
wnx\ case: & \bullet \varphi & \leftrightarrow & \mathbf{last} \vee \circ \varphi \\
snx\ case: & \circ \varphi & \leftrightarrow & \neg \mathbf{last} \wedge \bullet \varphi
\end{array}$$

The rules directly follow from the semantics of \circ (see Def. 13). \square

4.2.2 Converting normal form

With the help of the following rules, formulas can be converted from disjunctive to conjunctive normal form and vice versa. Rules *dnf lst* and *dnf stp* are applied to the different transitions of a formula in disjunctive normal form, rules *cnf lst* and *cnf stp* are applicable to the transitions of a conjunctive normal form.

Lemma 12 (*Conversion of normal forms*) *The following rules, which are used to convert disjunctive to conjunctive normal form and vice versa, are sound.*

$$\begin{array}{ll}
 \text{dnf lst:} & \rho \wedge \mathbf{last} \leftrightarrow (\rho \vee \neg \mathbf{last}) \\
 & \wedge (\mathbf{false} \vee \mathbf{false} \vee \mathbf{false} \vee \bullet \mathbf{false}) \\
 \text{dnf stp:} & \rho \wedge \delta \wedge \beta \wedge \circ \varphi \leftrightarrow (\rho \vee \mathbf{false} \vee \mathbf{false} \vee \bullet \mathbf{false}) \\
 & \wedge (\mathbf{false} \vee \delta \vee \mathbf{false} \vee \bullet \mathbf{false}) \\
 & \wedge (\mathbf{false} \vee \mathbf{false} \vee \beta \vee \bullet \mathbf{false}) \\
 & \wedge (\mathbf{false} \vee \mathbf{false} \vee \mathbf{false} \vee \circ \varphi) \\
 \text{cnf lst:} & (\rho \vee \neg \mathbf{last}) \leftrightarrow (\rho \wedge \mathbf{last}) \\
 & \vee (\mathbf{true} \wedge \mathbf{true} \wedge \mathbf{true} \wedge \circ \mathbf{true}) \\
 \text{cnf stp:} & (\rho \vee \delta \vee \beta \vee \bullet \varphi) \leftrightarrow (\rho \wedge \mathbf{true} \wedge \mathbf{true} \wedge \circ \mathbf{true}) \\
 & \vee (\mathbf{true} \wedge \delta \wedge \mathbf{true} \wedge \circ \mathbf{true}) \\
 & \vee (\mathbf{true} \wedge \mathbf{true} \wedge \beta \wedge \circ \mathbf{true}) \\
 & \vee (\mathbf{true} \wedge \mathbf{true} \wedge \mathbf{true} \wedge \bullet \varphi)
 \end{array}$$

We prove *cnf stp*. A proof for the other rules is similar. Trivial simplification of the right hand side of the equivalence (with rules *con true 1* and *con true 2*) gives

$$\begin{aligned}
 \vdash (\rho \vee \delta \vee \beta \vee \bullet \varphi) \leftrightarrow & \rho \wedge \circ \mathbf{true} \\
 & \vee \delta \wedge \circ \mathbf{true} \\
 & \vee \beta \wedge \circ \mathbf{true} \\
 & \vee \bullet \varphi
 \end{aligned}$$

If the current state is the last, $\bullet \varphi$ is satisfied and the equivalence trivially holds. Thus, we can assume that there is a next state. If there is a next state, then $\circ \mathbf{true} \leftrightarrow \mathbf{true}$. Rewriting with this equivalence and simplifying the result gives

$$\begin{aligned}
 \vdash (\rho \vee \delta \vee \beta \vee \bullet \varphi) \leftrightarrow & \rho \\
 & \vee \delta \\
 & \vee \beta \\
 & \vee \bullet \varphi
 \end{aligned}$$

□

4.3 Step

Assuming that the proof obligation has been rewritten to (strong conjunctive) normal form

$$\vdash (\rho_0 \vee \neg \mathbf{last}) \wedge \left(\bigwedge_{i=1}^n (\forall \vec{X}_i. \tau_i^d \vee \bullet \psi_i) \right),$$

we proceed as follows. Rule *con* is used to split up the different transitions, and rule *all* to eliminate the leading universal quantifiers.

$$\frac{\vdash (\rho_0 \vee \neg \mathbf{last}) \quad \frac{\vdash \tau_1^d \vee \bullet \psi_1}{\vdash \forall \vec{X}_1. \tau_1^d \vee \bullet \psi_1} \text{all} \quad \dots \quad \frac{\vdash \tau_n^d \vee \bullet \psi_n}{\vdash \forall \vec{X}_n. \tau_n^d \vee \bullet \psi_n} \text{all}}{\vdash (\rho_0 \vee \neg \mathbf{last}) \wedge \left(\bigwedge_{i=1}^n (\forall \vec{X}_i. \tau_i^d \vee \bullet \psi_i) \right)} \text{con}$$

Lemma 13 (see below) gives rules *lst* and *stp* which match the remaining premises. These calculus rules require the following auxiliary definition of functions *frm* and *frm^c* which are used to apply a frame assumption to a transition predicate ρ .

Definition 50 (*Application of frame assumption*) Applying a frame assumption δ to a given transition predicate ρ is defined as follows:

$$\begin{aligned} \text{frm}(\rho, \text{true}) &:= \rho \\ \text{frm}(\rho, \text{false}) &:= \text{false} \\ \text{frm}(\rho, [\mathbf{x}_1, \dots, \mathbf{x}_n]) &:= \rho[\mathbf{x}/\mathbf{x}'] \\ \text{frm}(\rho, \neg [\mathbf{x}_1, \dots, \mathbf{x}_n]) &:= \rho \\ \text{frm}^c(\rho, \text{true}) &:= \text{true} \\ \text{frm}^c(\rho, \text{false}) &:= \rho \\ \text{frm}^c(\rho, [\mathbf{x}_1, \dots, \mathbf{x}_n]) &:= \rho \\ \text{frm}^c(\rho, \neg [\mathbf{x}_1, \dots, \mathbf{x}_n]) &:= \rho[\mathbf{x}/\mathbf{x}'] \end{aligned}$$

where $\mathbf{x} := (\mathbf{Z} \cap \text{free}(\rho)) \setminus \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$

Lemma 13 (*Step rules*) Let $w = (\mathbf{Y} \cup \mathbf{Z}) \cap \text{free}(\rho)$. The following rules, which are used to execute a transition of a sequent, are sound.

$$\frac{\vdash \rho[\mathbf{X}, \mathbf{X}, \mathbf{X}/w, w', w'']}{\vdash \rho \vee \neg \mathbf{last}} \text{lst}$$

where \mathbf{X} fresh with respect to $\text{free}(\rho)$.

$$\frac{\vdash \text{frm}^c(\rho, \delta)[\mathbf{X}_1, \mathbf{X}_2, w/w, w', w''] \vee \varphi}{\vdash (\rho \vee \delta \vee \beta \vee \bullet \varphi)} \text{stp}$$

where $\mathbf{X}_1, \mathbf{X}_2$ fresh with respect to $\text{free}(\rho, \varphi)$.

If execution terminates, all free dynamic and program variables w – no matter, if they are unprimed, primed or double primed – are replaced by fresh static variables X . The result is a formula in pure predicate logic with static variables only, which can be proven with standard first order reasoning. Rule *stp* advances a step in the trace. The values of the dynamic and program variables w and w' in the old state are stored in fresh static variables X_1 and X_2 . Double primed variables are unprimed variables in the next state. Finally, the leading next operator of $\bullet \varphi$ is discarded. The proof method now continues with the execution of φ .

(Proof for Lemma 13) The proof for *stp* is more interesting than for *lst*. It mainly relies on Lemma 9; the valuation of a transition only depends on the first states $I(0)$, $I(0)'$, and $I(0)''$ – which corresponds to $I(1)$ – of an interval I . Thus, the interval can be divided as follows:

$$I = (I(0), I(0)', I(1)) + (I(1), \dots) ,$$

and the valuations of dynamic and program variables w in states $I(0)$ and $I(0)'$ can be “copied” to fresh static variables of $I(1)$. Copying of variables is possible because of the coincidence lemma (see Theorem 1). \square

4.4 Symbolic Execution

Symbolic execution is to rewrite a formula to normal form. The desired normal form depends on the position of the sub-formula. The top level formula must be rewritten to conjunctive normal form in order to apply rules *lst* and *stp* of Lemma 13. For an implication, the sub-formula on the left must be rewritten to disjunctive normal form instead. A disjunctive normal form can be converted into a conjunctive normal form and vice versa (see Lemma 12). Thus, for each operator, it is sufficient to provide rules to convert the operator into conjunctive *or* disjunctive normal form. Often, the disjunctive normal form is more intuitive.

We first consider sequential programs in Section 4.5. Program operators and especially sequential composition of programs are most suitable to explain the idea of symbolic execution. The idea is summarized in Sect. 4.6 where we informally classify the different operators and define a general pattern for the execution of certain types of operators. This pattern is used to define rewrite rules for the execution of interleaving in Section 4.7.1. Furthermore, local variable definition in Sect. 4.8 and procedure calls in Sect. 4.9 are discussed before an example for the execution of parallel programs is given in Section 4.10. Operators of temporal logic can also be “symbolically executed”. This is explained in Section 4.11 followed by Sections 4.12 and 4.13 where detailed rules for the propositional combination of normal forms are given. Another example in Section 4.14 illustrates how to execute temporal formulas. Finally, Section 4.16 concludes.

4.5 Sequential programs

4.5.1 Assignments

To illustrate our idea of symbolic execution, we first consider simple assignments $x := e$. An assignment takes exactly one step to assign to the program variable x (in the next state) the value of expression e (in the current state). All other variables are unchanged

$$x := e \quad \leftrightarrow \quad x' = e \wedge [x] \wedge \neg \mathbf{blocked} \wedge \circ \mathbf{last}$$

We refer to the value of the program variable x with a prime. The primed variable x' must be equal to e . The frame assumption $[x]$ ensures that all variables except x are unchanged. The formula $\circ \mathbf{last}$ states that there is a next step which is the last. Additionally, formula $\neg \mathbf{blocked}$ is used to express that an assignment does not block. This equivalence separates an assignment into a part defining the next transition ($x' = e \wedge [x] \wedge \neg \mathbf{blocked}$) and another part which is concerned with the rest of the trace ($\circ \mathbf{last}$).

We will use the equivalence above as a rule to rewrite an assignment to disjunctive normal form which we name *asg*. Similarly, other basic programs can be executed.

Lemma 14 (*Symbolic execution of assignments*) *The following rules, which are used to execute assignments, are sound.*

$$\begin{array}{lll} \mathit{asg}: & x := e & \leftrightarrow \quad x' = e \wedge [x] \wedge \neg \mathbf{blocked} \wedge \circ \mathbf{last} \\ \mathit{skp}: & \mathbf{skip} & \leftrightarrow \quad [] \wedge \neg \mathbf{blocked} \wedge \circ \mathbf{last} \\ \mathit{rnd}: & x := ? & \leftrightarrow \quad [x] \wedge \neg \mathbf{blocked} \wedge \circ \mathbf{last} \end{array}$$

Rule *asg* corresponds closely to the semantics of assignments.

$$\begin{array}{l} x := e \\ \equiv \quad x' = e \wedge [x] \wedge \circ \mathbf{last} \quad \text{Sem. } := \end{array}$$

The additional formula $\neg \mathbf{blocked}$ is equivalent to $\mathbf{blk}' = \mathbf{blk}$ by definition (see Def. 19). Because $x \neq \mathbf{blk}$, the equation follows from the frame assumption $[x]$.

Rules *skp* and *rnd* can be similarly established. □

4.5.2 Conditionals

The next construct we consider is a conditional

$$\mathbf{if} \ \psi \ \mathbf{then} \ \varphi_1 \ \mathbf{else} \ \varphi_2 \ .$$

Lemma 15 (*Symbolic execution of conditionals*) *The following rule, which is used to execute conditionals, is sound.*

$$\text{ite: } \mathbf{if } \psi \mathbf{ then } \varphi_1 \mathbf{ else } \varphi_2 \quad \leftrightarrow \quad \psi \wedge \varphi_1 \vee \neg \psi \wedge \varphi_2$$

The rule follows from Def. 16 with simple PL reasoning. \square

Rule *ite* gives two cases. In the first case, the condition ψ is true and we follow the then-branch of the program, in the second case the else-branch is considered as the condition is false. We have executed the conditional, but did not yet arrive with a formula which separates the first transition from the rest of the trace. We have to continue with the execution of either φ_1 or φ_2 .

Example Consider the program

$$\mathbf{if } n = 0 \mathbf{ then } m := 1 \mathbf{ else } m := 2.$$

In order to execute a step of the program, we first rewrite the conditional with rule *ite*.

$$n = 0 \wedge m := 1 \vee \neg n = 0 \wedge m := 2$$

Now we apply rule *asg* to rewrite the first assignment originating from the then-branch

$$\begin{aligned} & n = 0 \wedge m' = 1 \wedge [m] \wedge \neg \mathbf{blocked} \wedge \circ \mathbf{last} \\ & \vee \neg n = 0 \wedge m := 2 \end{aligned}$$

and also the second assignment of the else-branch.

$$\begin{aligned} & n = 0 \wedge m' = 1 \wedge [m] \wedge \neg \mathbf{blocked} \wedge \circ \mathbf{last} \\ & \vee \neg n = 0 \wedge m' = 2 \wedge [m] \wedge \neg \mathbf{blocked} \wedge \circ \mathbf{last} \end{aligned}$$

This leaves us with a disjunction of two formulas where the first transition changes m either to 1 or to 2. In both cases, the system terminates in the next step.

Alternative: If the semantics of conditionals is defined to require a step to evaluate the condition, the corresponding rewrite rule can be given as follows

$$\text{ite}^+: \mathbf{if}^+ \psi \mathbf{ then } \varphi_1 \mathbf{ else } \varphi_2 \quad \leftrightarrow \quad \begin{aligned} & \psi \wedge \square \wedge \neg \mathbf{blocked} \wedge \circ \varphi_1 \\ & \vee \neg \psi \wedge \square \wedge \neg \mathbf{blocked} \wedge \circ \varphi_2 \end{aligned}$$

This interpretation of a conditional is equivalent to a conditional which takes no time to evaluate its condition but contains an extra **skip** statement in each branch:

$$\mathbf{if}^+ \psi \mathbf{ then } \varphi_1 \mathbf{ else } \varphi_2 \quad := \quad \mathbf{if } \psi \mathbf{ then } (\mathbf{skip}; \varphi_1) \mathbf{ else } (\mathbf{skip}; \varphi_2) .$$

4.5.3 Loops

A while loop

while ψ **do** φ

is rewritten to an equivalent formula as follows.

Lemma 16 (*Symbolic execution of while loops*) *The following rule, which is used to execute while loops, is sound.*

$$\text{whl: } \mathbf{while} \ \psi \ \mathbf{do} \ \varphi \quad \leftrightarrow \quad \begin{array}{l} (\psi \wedge \varphi \wedge \circ \text{true}); \mathbf{while} \ \psi \ \mathbf{do} \ \varphi \\ \vee \neg \psi \wedge \mathbf{last} \end{array}$$

In the first case where condition ψ is true, the body of the while loop is executed followed by a repeated execution of the while loop. In the second case where condition ψ is false, the while loop terminates. Note that the execution of body φ must take at least one step as otherwise the semantics of the while loop is not well-defined.

(Proof for Lemma 16) The proof follows from property *chopstareqv* of operator φ^* [7]:

$$\varphi^* \quad \leftrightarrow \quad \mathbf{last} \vee (\varphi \wedge \circ \text{true}); \varphi^*$$

□

Alternative: If the semantics of while loops is defined to require a step to evaluate the condition, the corresponding rewrite rule can be given as follows

$$\text{whl}^+: \quad \mathbf{while}^+ \ \psi \ \mathbf{do} \ \varphi \quad \leftrightarrow \quad \begin{array}{l} \psi \wedge \square \wedge \neg \mathbf{blocked} \wedge \circ (\varphi; \mathbf{while}^+ \ \psi \ \mathbf{do} \ \varphi) \\ \vee \neg \psi \wedge \square \wedge \neg \mathbf{blocked} \wedge \circ \mathbf{last} \end{array}$$

Again, this interpretation of a while loop is equivalent to a while loop which takes no time to evaluate its condition but contains extra **skip** statements:

$$\mathbf{while}^+ \ \psi \ \mathbf{do} \ \varphi \quad \equiv \quad (\mathbf{while} \ \psi \ \mathbf{do} \ (\mathbf{skip}; \varphi)); \mathbf{skip} .$$

4.5.4 Sequential composition

The execution of sequential composition of programs $\varphi; \psi$ is more complicated as we cannot give a simple equivalence which rewrites a composition to normal form. The problem is that the first formula φ could take an unknown number of steps to execute. Only after φ has terminated, we continue with executing ψ .

In order to execute composition, the idea is to first rewrite formula φ to normal form

$$\left(\rho_0 \wedge \mathbf{last} \vee \left(\bigvee (\exists \vec{X}_i. \tau_i \wedge \circ \varphi_i) \right) \right); \psi$$

The first sub-formula φ can be rewritten, because sequential composition adheres to the following property:

$$\text{chp lem: } [\text{true}] (\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1; \psi \rightarrow \varphi_2; \psi)$$

If φ_1 implies φ_2 on all continuing paths, then $\varphi_1; \psi$ also implies $\varphi_2; \psi$. This rule can be turned into congruence rules according to Appendix A.5.2.

After rewriting the first sub-formula to normal form, we rewrite the composition operator. Composition adheres to the following property

$$\text{chp dis: } (\varphi_1 \vee \varphi_2); \psi \leftrightarrow \varphi_1; \psi \vee \varphi_2; \psi$$

i.e., composition distributes over disjunction. If we apply this property to the formula above, we receive a number of cases

$$(\rho_0 \wedge \mathbf{last}); \psi \vee \left(\bigvee (\exists \vec{X}_i. \tau_i^c \wedge \circ \varphi_i); \psi \right)$$

Composition also distributes over existential quantification

$$\text{chp ex: } (\exists v. \varphi); \psi \leftrightarrow \exists v_0. \varphi[v_0/v]; \psi$$

leading to

$$(\rho_0 \wedge \mathbf{last}); \psi \vee \left(\bigvee \exists \vec{X}_{i,0}. (\tau_{i,0}^c \wedge \circ \varphi_{i,0}); \psi \right)$$

In the first case, program φ terminates, in the other cases, the program takes a step τ_i and continues with program ψ_i . Now we use the following equivalences

$$\begin{aligned} \text{chp lst: } (\rho \wedge \mathbf{last}); \psi &\leftrightarrow \rho[w,w'/w'',w''] \wedge \psi \\ \text{chp stp: } (\tau^c \wedge \circ \varphi); \psi &\leftrightarrow \tau^c \wedge \circ (\varphi; \psi) \end{aligned}$$

to further rewrite the composition. Dynamic and program variables w stutter in the last step, and therefore the primed and double primed variables w' and w'' of ρ are replaced by the corresponding unprimed variables if the first sub-formula terminates. The two rules give

$$\rho_0[w,w'/w'',w''] \wedge \psi \vee \left(\bigvee \exists \vec{X}_{i,0}. \tau_{i,0}^c \wedge \circ (\varphi_{i,0}; \psi) \right)$$

In the first case, φ has terminated and we still need to execute ψ to arrive with a formula in normal form. In the other cases, we have successfully separated the formula into the first transition $\tau_{i,0}^c$ and the corresponding rest of the program $\varphi_{i,0}; \psi$.

Lemma 17 (*Symbolic execution of sequential composition*) *The following set of rules, which are used to rewrite a composition $\varphi; \psi$ to disjunctive normal form, are sound.*

$$\begin{aligned}
\text{chp lem: } & [\text{true}] (\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1; \psi \\
& \qquad \qquad \qquad \rightarrow \varphi_2; \psi) \\
\text{chp dis: } & (\varphi_1 \vee \varphi_2); \psi \leftrightarrow \varphi_1; \psi \vee \varphi_2; \psi \\
\text{chp ex: } & (\exists v. \varphi); \psi \leftrightarrow \exists v_0. \varphi[v_0/v]; \psi \\
& \qquad v_0 \text{ fresh with respect to } (\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(\psi) \\
\text{chp lst: } & (\rho \wedge \mathbf{last}); \psi \leftrightarrow \rho^{[w,w/w',w'']} \wedge \psi \\
\text{chp stp: } & (\tau^c \wedge \circ \varphi); \psi \leftrightarrow (\tau^c \wedge \circ (\varphi; \psi))
\end{aligned}$$

See Appendix C.2.1. □

Example Consider program

$$(n := 0; m := 1); n := 1$$

which is a sequence of assignments. In order to rewrite the outermost composition, we need to rewrite the first sub-program which again is a composition. With the help of property *chp lem* we can rewrite the first assignment below the sequential compositions with rule *asg*.

$$((n' = 0 \wedge [n] \wedge \neg \mathbf{blocked} \wedge \circ \mathbf{last}); m := 1); n := 1$$

The first assignment is now in normal form. It contains one case which describes the only possible transition. We use *chp stp* to rewrite the inner composition

$$(n' = 0 \wedge [n] \wedge \neg \mathbf{blocked} \wedge \circ (\mathbf{last}; m := 1)); n := 1$$

We again apply *chp stp* to rewrite the outermost composition to

$$n' = 0 \wedge [n] \wedge \neg \mathbf{blocked} \wedge \circ ((\mathbf{last}; m := 1); n := 1)$$

Now the whole formula has been rewritten to an equivalent formula in normal form. In the first step 0 is assigned to variable n . All other variables are unchanged.

In the next step, the first assignment will terminate

$$(\mathbf{last}; m := 1); n := 1$$

Here, we apply rule *chp lst* to the inner composition leading to

$$(\text{true} \wedge m := 1); n := 1$$

which can be simplified to

$$m := 1; n := 1$$

Execution continues with rewriting the next assignment $m := 1$.

4.6 Excursion: Classification of operators

Before continuing to present rules for symbolic execution of operators, we would like to take a step back to discuss a pattern behind rules. Some operators can be directly rewritten. For example assignments $x := e$, but also conditionals **if** φ **then** ψ_1 **else** ψ_2 or while loops **while** φ **do** ψ . For other operators the sub-formulas must be rewritten to normal form first. For the latter, the calculus rules always follow a certain pattern.

In general, if we encounter an operator \oplus which cannot be directly rewritten to normal form, we rewrite the sub-formulas first and afterwards execute the operator itself. In this case, we try to define calculus rules according to the following schemes.

Definition 51 (*Pattern for executing existential operators*)

$$\begin{aligned}
\oplus \text{ lem: } & \dots \rightarrow (\oplus(\varphi_1) \rightarrow \oplus(\varphi_2)) \\
\oplus \text{ dis: } & \oplus(\varphi_1 \vee \varphi_2) \leftrightarrow \oplus(\varphi_1) \vee \oplus(\varphi_2) \\
\oplus \text{ ex: } & \oplus(\exists v. \varphi) \leftrightarrow \exists v_0. \oplus(\varphi[v_0/v]) \\
& v_0 \text{ fresh with respect to } \text{free}(\oplus(\exists v. \varphi)) \\
\oplus \text{ lst: } & \oplus(\rho \wedge \mathbf{last}) \leftrightarrow \dots \\
\oplus \text{ stp: } & \oplus(\tau^c \wedge \circ \varphi) \leftrightarrow \dots
\end{aligned}$$

For operators $\varphi; \psi$, $l_1 :: \varphi \parallel^< l_2 :: \psi$, $\varphi \parallel \psi$, $\exists x. \varphi$, $\{\varphi\}$, and $\langle \varphi \rangle \psi$ corresponding rules can be defined. Operators of this category are also called existential operators. If there are more than one sub-formula to an operator, rules are more complex.

There are operators which do not distribute over disjunction, but rather distribute over conjunction and are therefore classified as a universal operators.

Definition 52 (*Pattern for executing universal operators*)

$$\begin{aligned}
\otimes \text{ lem: } & \dots \rightarrow (\otimes(\varphi_1) \rightarrow \otimes(\varphi_2)) \\
\otimes \text{ con: } & \otimes(\varphi_1 \wedge \varphi_2) \leftrightarrow \otimes(\varphi_1) \wedge \otimes(\varphi_2) \\
\otimes \text{ all: } & \otimes(\forall v. \varphi) \leftrightarrow \forall v_0. \otimes(\varphi[v_0/v]) \\
& v_0 \text{ fresh with respect to } \text{free}(\otimes(\forall v. \varphi)) \\
\otimes \text{ lst: } & \otimes(\rho \vee \neg \mathbf{last}) \leftrightarrow \dots \\
\otimes \text{ stp: } & \otimes(\tau^d \vee \bullet \varphi) \leftrightarrow \dots
\end{aligned}$$

Operator $\forall x. \varphi$ falls into this category. Operator $[\varphi] \psi$ distributes over disjunction for its first argument φ , and over conjunction for its second argument ψ .

Boolean operators can also be classified accordingly. While the boolean operator $\varphi \wedge \psi$ distributes over disjunction, $\varphi \vee \psi$ distributes over conjunction, and $\varphi \rightarrow \psi$ is existential in its first argument and universal in its second. Negation $\neg \varphi$ distributes both over

disjunction and conjunction and can be seen as an existential and universal operator. Equivalence $\varphi \leftrightarrow \psi$ is neither existential nor universal in both arguments, but can be rewritten to $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$.

The turn-style \vdash can also be seen as a universal operator. The rules *rewrite*, *con*, *all* (Section 3.2.2), *lst*, *stp* (Lemma 13) follow the style of Definition 52.

4.7 Parallel programs

4.7.1 Interleaving

As with sequential composition, the interleaving of programs cannot be executed directly, but the sub-formulas need to be rewritten to normal form first. The basic operator for interleaving is the left interleaving operator $l_1 :: \varphi \parallel^< l_2 :: \psi$ which gives precedence to the left process. Other operators are defined as abbreviations as follows (compare to Def. 24):

$$\begin{aligned} l_1 :: \varphi \parallel^> l_2 :: \psi &::= l_2 :: \psi \parallel^< l_1 :: \varphi \\ l_1 :: \varphi \parallel l_2 :: \psi &::= l_1 :: \varphi \parallel^< l_2 :: \psi \vee l_1 :: \varphi \parallel^> l_2 :: \psi \\ l_1 :: \varphi \parallel_b^< l_2 :: \psi &::= l_1 :: (\mathbf{blocked} \wedge \circ \varphi) \parallel^< l_2 :: \psi \\ l_1 :: \varphi \parallel_b^> l_2 :: \psi &::= l_1 :: \varphi \parallel^> l_2 :: (\mathbf{blocked} \wedge \circ \psi) \end{aligned}$$

According to this, the interleaving operator can simply be rewritten to a left and a right interleaving operator.

$$ilv: \quad l_1 :: \varphi \parallel l_2 :: \psi \quad \leftrightarrow \quad l_1 :: \varphi \parallel^< l_2 :: \psi \vee l_1 :: \varphi \parallel^> l_2 :: \psi$$

Either the first or the second process takes precedence. In order to execute $\parallel^<$, the first sub-formula must be rewritten to normal form before the operator itself can be rewritten. Left interleaving distributes over disjunction and the following rewrite rules hold.

Lemma 18 (*Symbolic execution of left interleaving*) *The following set of rules, which are used to rewrite left interleaving $l_1 :: \varphi \parallel^< l_2 :: \psi$ to disjunctive normal form, are sound.*

$$\begin{aligned} ilvl \text{ lem:} \quad & [\text{true}] (\varphi_1 \rightarrow \varphi_2) \rightarrow (\quad l_1 :: \varphi_1 \parallel^< l_2 :: \psi \\ & \quad \rightarrow l_1 :: \varphi_2 \parallel^< l_2 :: \psi) \\ ilvl \text{ dis:} \quad & l_1 :: (\varphi_1 \vee \varphi_2) \parallel^< l_2 :: \psi \\ & \leftrightarrow l_1 :: \varphi_1 \parallel^< l_2 :: \psi \vee l_1 :: \varphi_2 \parallel^< l_2 :: \psi \\ ilvl \text{ ex:} \quad & l_1 :: (\exists v. \varphi) \parallel^< l_2 :: \psi \\ & \leftrightarrow \exists v_0. l_1 :: \varphi^{[v_0/v]} \parallel^< l_2 :: \psi \\ & \quad v_0 \text{ fresh with respect to } (\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(l_1, l_2, \psi) \\ ilvl \text{ lst:} \quad & l_1 :: (\rho \wedge \mathbf{last}) \parallel^< l_2 :: \psi \quad \leftrightarrow \quad \rho^{[w,w/w',w'']} \wedge \psi \\ ilvl \text{ stp:} \quad & l_1 :: (\rho \wedge \delta \wedge \neg \mathbf{blocked} \wedge \circ \varphi) \parallel^< l_2 :: \psi \\ & \leftrightarrow \exists X_2. (\quad (\rho^{[X_2/w'']} \wedge \neg \text{trm}(l_2)) \wedge \delta \wedge \neg \mathbf{blocked} \\ & \quad \wedge \circ (l_1 :: (w = X_2 \wedge \varphi) \parallel l_2 :: \psi)) \end{aligned}$$

$$\begin{aligned}
\text{\textit{ilvlb lem}:} & \quad [\text{true}] (\varphi_1 \rightarrow \varphi_2) \rightarrow (\quad l_2 :: \psi \parallel_b^< l_1 :: \varphi_1 \\
& \quad \rightarrow l_2 :: \psi \parallel_b^< l_1 :: \varphi_2) \\
\text{\textit{ilvlb dis}:} & \quad l_2 :: \psi \parallel_b^< l_1 :: (\varphi_1 \vee \varphi_2) \\
& \quad \leftrightarrow l_2 :: \psi \parallel_b^< l_1 :: \varphi_1 \vee l_2 :: \psi \parallel_b^< l_1 :: \varphi_2 \\
\text{\textit{ilvlb ex}:} & \quad l_2 :: \psi \parallel_b^< l_1 :: (\exists v. \varphi) \\
& \quad \leftrightarrow \exists v_0. l_2 :: \psi \parallel_b^< l_1 :: \varphi^{[v_0/v]} \\
& \quad v_0 \text{ fresh with respect to } (\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(l_1, l_2, \psi) \\
\text{\textit{ilvlb lst}:} & \quad l_2 :: \psi \parallel_b^< l_1 :: (\rho \wedge \mathbf{last}) \quad \leftrightarrow \quad \text{false} \\
\text{\textit{ilvlb stp}:} & \quad l_2 :: \psi \parallel_b^< l_1 :: (\rho \wedge \delta \wedge \beta \wedge \circ \varphi) \\
& \quad \leftrightarrow \exists X_{2,2}. (\quad \rho^{[X_{2,2}/w'']} \wedge \delta \wedge \beta \\
& \quad \wedge \circ (l_2 :: \psi \parallel l_1 :: (w = X_{2,2} \wedge \varphi)))
\end{aligned}$$

The operator $\varphi \parallel_b^< \psi$ is existential in its second argument. The second process must not terminate (*ilvlb lst*), because this case is already considered in $\varphi \parallel_b^> \psi$ which gives precedence to the second process (compare Section 2.4.3). A transition of the second process is executed with rule *ilvlb stp* no matter if the transition is blocked. If it is blocked, then the overall interleaving is also blocked. Otherwise, the rule is similar to rule *ilvl stp*. All-together, transitions of both processes have been executed. The right hand side of Fig. 4.1 illustrates how static variables have been introduced for the primed and double primed variables of the different processes: the primed variables of the first process have been replaced by X_1 as they do not contribute to the overall transition. Double primed variables of both processes have been replaced by $X_{2,1}$ and $X_{2,2}$ to ensure that the environment transitions of the two processes include the transitions of the other process.

(Proof of Lemmas 18 and 19) It is surprisingly straightforward to verify *ilvl lem*, *ilvl dis*, *ilvl ex*, *ilvlb lem*, *ilvlb dis*, and *ilvlb ex*. Rules *ilvl lst* and *ilvl stp* directly correspond to the SOS rules of Def. 24 with the same name. Together, the rules *ilvl blk* and *ilvlb stp* “implement” the SOS rule *ilvl blk*. \square

Alternative: Interleaving can be executed without the use of the operators $\parallel_b^<$ and $\parallel_b^>$. The idea is to rewrite both sub-formulas to normal form and then to apply a rewrite rule depending on the sub formulas being blocked. An appropriate rule where both sub-formulas progress is as follows.

$$\begin{aligned}
\text{\textit{ilv stp stp}:} & \quad (\rho_1 \wedge \delta_1 \wedge \neg \mathbf{blocked} \wedge \circ \varphi_1) \\
& \quad \parallel (\rho_2 \wedge \delta_2 \wedge \neg \mathbf{blocked} \wedge \circ \varphi_2) \\
& \quad \leftrightarrow \exists X_2. (\quad \rho_1^{[X_2/w'']} \wedge \delta_1 \wedge \neg \mathbf{blocked} \\
& \quad \wedge \circ ((w = X_2 \wedge \varphi_1) \parallel (\rho_2 \wedge \delta_2 \wedge \neg \mathbf{blocked} \wedge \circ \varphi_2))) \\
& \quad \vee \exists X_2. (\quad \rho_2^{[X_2/w'']} \wedge \delta_2 \wedge \neg \mathbf{blocked} \\
& \quad \wedge \circ ((\varphi_1 \wedge \delta_1 \wedge \neg \mathbf{blocked} \wedge \circ \varphi_1) \parallel (w = X_2 \wedge \varphi_2)))
\end{aligned}$$

In the two cases of the result, either the transition of the first or the second sub-formula is executed; the other formula is unchanged. In practice, it turns out to be disadvantageous that a formula has been rewritten to normal form even if it is not executed.

4.7.2 Synchronization

Lemma 20 (*Symbolic execution of synchronization*) *The following rule, which is used to rewrite synchronization **await** φ to disjunctive normal form, is sound.*

$$\text{awt: } \mathbf{await} \ \varepsilon \quad \leftrightarrow \quad \begin{array}{l} \varepsilon \wedge \mathbf{last} \\ \vee \quad \neg \varepsilon \wedge \mathbf{[]} \wedge \mathbf{blocked} \\ \wedge \circ \mathbf{await} \ \varepsilon \end{array}$$

Because **await** φ is defined as an abbreviation of a **while** loop (see Def. 19), the rule can be derived from Lemma 16. \square

Rule *awt* illustrates how synchronization behaves. If the condition is satisfied, the operator immediately terminates; evaluation of the condition does not take time. If the condition is false, the transition is blocked and the **await** statement iterates in the next step. It is important to note that this rule is only sound because the condition ε is a dynamic expression which is compliant to Def. 30. For the most general case, where the condition can be an arbitrary temporal formula, execution is a bit more complicated. In practice, the condition always is a dynamic expression and therefore, the general case has not been considered.

4.8 Local variables and quantifiers

It is possible to quantify program variables, e.g., operator $\exists x. \varphi$ hides a local variable from the outside world. This operator is comparable to the hiding operator of TLA. However, in TLA the semantics is defined modulo stuttering transitions while in our semantics, the environment is modeled with double primed variables. The proof method of TLA requires instances given as a so called state formula which defines the value of the variable in every state, i.e., an instance must be given in advance *for every future state*. Not surprisingly, the correct instance is often difficult to find. Here, we follow the idea of instantiating the hidden variable on the fly, i.e., to give instances for the current state only. The operator is an existential operator distributing over disjunction. Therefore, the set of calculus rules is as follows.

Lemma 21 (*Symbolic execution of hiding operator*) *The following rules, which are used to rewrite the hiding operator $\exists x. \varphi$ to disjunctive normal form, is sound.*

$$\begin{array}{l} \text{exx lem:} \quad (\forall x. \varphi_1 \rightarrow \varphi_2) \rightarrow (\exists x. \varphi_1 \rightarrow \exists x. \varphi_2) \\ \text{exx dis:} \quad (\exists x. \varphi_1 \vee \varphi_2) \leftrightarrow (\exists x. \varphi_1) \vee (\exists x. \varphi_2) \\ \text{exx ex:} \quad (\exists x. \exists v. \varphi) \leftrightarrow \exists v. \exists x. \varphi \\ \text{exx lst:} \quad (\exists x. \rho \wedge \mathbf{last}) \\ \quad \leftrightarrow (\exists X_0. \rho[X_0, X_0, X_0/x, x', x'']) \wedge \mathbf{last} \end{array}$$

$$\begin{array}{l}
X_0 \text{ fresh with respect to } \text{free}(\rho) \\
\text{exx stp:} \quad (\exists x. \rho \wedge \delta \wedge \beta \wedge \circ \varphi) \\
\quad \leftrightarrow \exists X_2. (\exists X_0, X_1. \rho_0) \wedge \delta_0 \wedge \beta \wedge \circ \exists x. x = X_2 \wedge \varphi \\
\rho_0 := \text{frm}(\rho, \delta)[^{X_0, X_1, X_2}_{/x, x', x''}] \\
\delta_0 := \begin{cases} [\vec{x} \cup \{x\}] & \delta \equiv [\vec{x}] \\ X_1 \neq X_0 \vee \neg [\vec{x} \cup \{x\}] & \delta \equiv \neg [\vec{x}], x \notin \vec{x} \\ \delta & \text{otherwise} \end{cases} \\
X_0, X_1, X_2 \text{ fresh with respect to } \text{free}(\rho, \varphi)
\end{array}$$

If the system terminates, variable x can be replaced by a single static variable X_0 (rule *exx lst*). If the system steps, three additional static variables are introduced for the unprimed, primed, and double primed variable x (rule *exx stp*). Thus, it is sufficient to provide a “local” instance of x for the current transition alone. In practice, the “local” instance can often be automatically determined.

Rules for the universal operator $\forall y. \varphi$ are very similar and can be found in Appendix B.7.2. With the existential hiding operator local variables can be defined in a parallel program with **var** $x = e$ **in** φ (see Section. 16). Derived rules for executing a local variable definition are listed in Appendix B.8.4.

4.9 Procedure calls

For a procedure, we need to distinguish between *calling* and *implementing* a procedure. For a procedure call, two basic properties can be derived for the value and the var parameters.

Lemma 22 (*Properties of procedure calls*)

$$\begin{array}{l}
\text{call lem val } i: \quad e_i^1 = e_i^2 \rightarrow (\quad \text{proc}(e_1, \dots, e_i^1, \dots, e_n; x_1, \dots, x_m) \\
\quad \quad \quad \leftrightarrow \text{proc}(e_1, \dots, e_i^2, \dots, e_n; x_1, \dots, x_m)) \\
\text{call frm:} \quad \text{proc}(e_1, \dots, e_n; x_1, \dots, x_m) \rightarrow \square [x_1, \dots, x_m, \text{blk}]
\end{array}$$

The two properties directly follow from the semantics of procedures (see Def. 17). \square

If – in the current state – the expression e_i^1 for a value parameter is equal to an expression e_i^2 , then also the procedure call is equivalent to a call where the value parameter has been replaced (*call lem val i*). Furthermore, all program variables except the var parameters and variable blk are unchanged in a system transition (*call frm*).

In order to “implement” a procedure, it can be defined to be equivalent to an arbitrary temporal formula. However, it must be ensured that the equivalence does not contradict the properties of the lemma above.

Example The equivalence

$$\text{send}(d; \mathbf{var} \ c) \leftrightarrow d := D$$

is not valid, because the assignment to variable d contradicts the frame assumption $[c, \text{blk}]$ of rule *call frm*; only the program variables which are listed as \mathbf{var} parameters can change.

The equivalence

$$\text{send}(d; \mathbf{var} \ c) \leftrightarrow \mathbf{skip}; c.\text{data} := d$$

is also invalid, as the validity of the right hand side depends on the value of variable d in the second state. However, procedure send only depends on the valuation of value parameters in the first state (compare to rule *call lem val i*).

In order to ensure consistency of procedures, we give an implementation of procedures as follows:

$$\text{proc}(v_1, \dots, v_n; x_1, \dots, x_m) \\ \alpha$$

Variables $v_1, \dots, v_n, x_1, \dots, x_m$ must all be different. Symbol α represents the body of the procedure. Static analysis can ensure that α is consistent and only refers to program variables v_1, \dots, v_n and x_1, \dots, x_m . The implementation can then be turned into a rewrite rule

$$\text{proc: } \text{proc}(v_1, \dots, v_n; x_1, \dots, x_m) \leftrightarrow \mathbf{var} \ v_1 = v_1, \dots, v_n = v_n \ \mathbf{in} \ \alpha$$

where v_1, \dots, v_n are fresh variables $\in \mathbf{Y}$. The additional local variable definition ensures that the value parameters are evaluated in the first state and that the global value of variables v_1, \dots, v_n is unchanged.

Example An implementation of procedure send

$$\text{send}(d; \mathbf{var} \ c) \\ \mathbf{await} \ c.\text{sig} = c.\text{ack}; \\ c.\text{data} := d; \\ c.\text{sig} := \neg c.\text{sig}$$

is turned into the following rewrite rule:

$$\text{send}(d_0; \mathbf{var} \ c) \\ \leftrightarrow \mathbf{var} \ d = d_0 \ \mathbf{in} \\ \mathbf{await} \ c.\text{sig} = c.\text{ack}; \\ c.\text{data} := d; \\ c.\text{sig} := \neg c.\text{sig}$$

An implementation ensures consistency of the specification and the derived rewrite rule is used to replace a procedure call with its implementation.

4.10 Example: Executing parallel programs

Example Consider program

$$\text{send}(d; \mathbf{var} \ c) \parallel \text{receive}(\mathbf{var} \ c, d) . \quad (4.1)$$

which we would like to execute by translating the formula into disjunctive normal form. Rule *ilv* gives two cases

$$\begin{aligned} & \text{send}(d; \mathbf{var} \ c) \parallel^< \text{receive}(\mathbf{var} \ c, d) \\ \vee & \text{send}(d; \mathbf{var} \ c) \parallel^> \text{receive}(\mathbf{var} \ c, d) . \end{aligned}$$

In the first case, the procedure `send` can be replaced by its implementation.

$$\left(\begin{array}{l} \mathbf{var} \ d = d \ \mathbf{in} \\ \quad \mathbf{await} \ c.\text{sig} = c.\text{ack}; \\ \quad c.\text{data} := d; \\ \quad c.\text{sig} := \neg c.\text{sig} \end{array} \right) \parallel^< \text{receive}(\mathbf{var} \ c, d) .$$

Rewriting `await` with property *awt* leads to

$$\left(\begin{array}{l} \mathbf{var} \ d = d \ \mathbf{in} \\ \quad \left(\begin{array}{l} c.\text{sig} = c.\text{ack} \wedge \mathbf{last} \\ \vee \neg c.\text{sig} = c.\text{ack} \wedge \square \wedge \mathbf{blocked} \end{array} \right); \\ \quad \wedge \circ \mathbf{await} \ c.\text{sig} = c.\text{ack} \\ \quad c.\text{data} := d; \\ \quad c.\text{sig} := \neg c.\text{sig} \end{array} \right) \parallel^< \text{receive}(\mathbf{var} \ c, d) .$$

The disjunction is lifted to top-level with the successive application of rules *chp dis*, *var dis*, and *ilvl dis*

$$\begin{aligned} & \left(\begin{array}{l} \mathbf{var} \ d = d \ \mathbf{in} \\ \quad (c.\text{sig} = c.\text{ack} \wedge \mathbf{last}); \\ \quad c.\text{data} := d; \\ \quad c.\text{sig} := \neg c.\text{sig} \end{array} \right) \parallel^< \text{receive}(\mathbf{var} \ c, d) \\ \vee & \left(\begin{array}{l} \mathbf{var} \ d = d \ \mathbf{in} \\ \quad \left(\begin{array}{l} \neg c.\text{sig} = c.\text{ack} \wedge \square \wedge \mathbf{blocked} \end{array} \right); \\ \quad \wedge \circ \mathbf{await} \ c.\text{sig} = c.\text{ack} \\ \quad c.\text{data} := d; \\ \quad c.\text{sig} := \neg c.\text{sig} \end{array} \right) \parallel^< \text{receive}(\mathbf{var} \ c, d) \end{aligned}$$

leading to two additional top level cases. In the first case, rule *chp lst* gives the following result:

$$\left(\begin{array}{l} \mathbf{var} \ d = d \ \mathbf{in} \\ \quad (c.\text{sig} = c.\text{ack} \wedge c.\text{data} := d); \\ \quad c.\text{sig} := \neg c.\text{sig} \end{array} \right) \parallel^< \text{receive}(\mathbf{var} \ c, d) .$$

The assignment $c.data := d$ is to be executed next. Rewriting the assignment with rule *asg* gives

$$\left(\begin{array}{l} \mathbf{var} \ d = d \ \mathbf{in} \\ \left(\begin{array}{l} c.sig = c.ack \wedge c' = c.setdata(c, d) \\ \wedge [c] \wedge \neg \mathbf{blocked} \\ \wedge \circ \mathbf{last} \end{array} \right); \end{array} \right) \parallel^< \text{receive}(\mathbf{var} \ c, d) .$$

The first formula of the sequential composition of the left process is in normal form. Now, rule *chp stp* can be applied to the composition.

$$\left(\begin{array}{l} \mathbf{var} \ d = d \ \mathbf{in} \\ c.sig = c.ack \wedge c' = c.setdata(c, d) \\ \wedge [c] \wedge \neg \mathbf{blocked} \\ \wedge \circ \left(\begin{array}{l} \mathbf{last}; \\ c.sig := \neg c.sig \end{array} \right) \end{array} \right) \parallel^< \text{receive}(\mathbf{var} \ c, d)$$

Rewriting the local variable definition with *var stp* leads to

$$\left(\begin{array}{l} c.sig = c.ack \wedge c' = c.setdata(c, d) \\ \wedge [c] \wedge \neg \mathbf{blocked} \\ \wedge \circ \left(\begin{array}{l} \mathbf{var} \ d = d \ \mathbf{in} \\ \mathbf{last}; \\ c.sig := \neg c.sig \end{array} \right) \end{array} \right) \parallel^< \text{receive}(\mathbf{var} \ c, d) ,$$

and finally rewriting the left interleaving with *ilvl stp* results in the transition

$$\begin{array}{l} c.sig = c.ack \wedge c' = c.setdata(c, d) \\ \wedge [c] \wedge \neg \mathbf{blocked} \\ \wedge \circ \left(\left(\begin{array}{l} \mathbf{var} \ d = d \ \mathbf{in} \\ \mathbf{last}; \\ c.sig := \neg c.sig \end{array} \right) \parallel \text{receive}(\mathbf{var} \ c, d) \right) . \end{array}$$

This transition is one out of six which result from executing the first step of program (4.1). All transitions are given in Figure 4.2. Note that the PL formulas of transition (4.4) and (4.7) are contradictory; simplification would eliminate the two. Also, because receive_b is equivalent to $\text{receive}(\mathbf{var} \ c, d)$, the transitions (4.2) and (4.6) are identical and can be contracted. The same holds true for transitions (4.3) and (4.5), because send_b is equivalent to $\text{send}(d; \mathbf{var} \ c)$. Only two transitions remain. This illustrates that automatic simplification during symbolic executing and also the use of sequencing to contract premises (see Chapter 5) is very important.

4.11 Temporal logic operators

The idea of symbolic execution can also be applied to formulas in temporal logic. For example, operator $\square \varphi$ is similar to a loop in a programming language: formula φ is executed in every step. An appropriate execution rule is

$$\begin{aligned} & c.\text{sig} = c.\text{ack} \wedge c' = c.\text{setdata}(c, d) \\ \wedge [c] \wedge \neg \mathbf{blocked} \wedge \circ (\text{send}_1 \parallel \text{receive}(\mathbf{var} c, d)) \end{aligned} \quad (4.2)$$

$$\begin{aligned} & \neg c.\text{sig} = c.\text{ack} \wedge c.\text{sig} \neq c.\text{ack} \wedge d' = c.\text{data} \\ \wedge [d] \wedge \neg \mathbf{blocked} \wedge \circ (\text{send}_b \parallel \text{receive}_1) \end{aligned} \quad (4.3)$$

$$\begin{aligned} & \neg c.\text{sig} = c.\text{ack} \wedge \neg c.\text{sig} \neq c.\text{ack} \\ \wedge [] \wedge \mathbf{blocked} \wedge \circ (\text{send}_b \parallel \text{receive}_b) \end{aligned} \quad (4.4)$$

$$\begin{aligned} & c.\text{sig} \neq c.\text{ack} \wedge d' = c.\text{data} \\ \wedge [d] \wedge \neg \mathbf{blocked} \wedge \circ (\text{send}(d; \mathbf{var} c) \parallel \text{receive}_1) \end{aligned} \quad (4.5)$$

$$\begin{aligned} & \neg c.\text{sig} \neq c.\text{ack} \wedge c.\text{sig} = c.\text{ack} \wedge c' = c.\text{setdata}(c, d) \\ \wedge [c] \wedge \neg \mathbf{blocked} \wedge \circ (\text{send}_1 \parallel \text{receive}_b) \end{aligned} \quad (4.6)$$

$$\begin{aligned} & \neg c.\text{sig} \neq c.\text{ack} \wedge \neg c.\text{sig} = c.\text{ack} \\ \wedge [] \wedge \mathbf{blocked} \wedge \circ (\text{send}_b \parallel \text{receive}_b) \end{aligned} \quad (4.7)$$

where

$$\begin{array}{ll} \text{send}_b & := \mathbf{var} d = d \mathbf{in} & \text{receive}_b & := \mathbf{await} c.\text{sig} \neq c.\text{ack}; \\ & \mathbf{await} c.\text{sig} = c.\text{ack}; & & d := c.\text{data}; \\ & c.\text{data} := d; & & c.\text{ack} := \neg c.\text{sig} \\ & c.\text{sig} := \neg c.\text{sig} & & \\ \\ \text{send}_1 & := \mathbf{var} d = d \mathbf{in} & \text{receive}_1 & := \mathbf{last}; \\ & \mathbf{last}; & & c.\text{ack} := \neg c.\text{sig} \\ & c.\text{sig} := \neg c.\text{sig} & & \end{array}$$

Figure 4.2: Complete set of example transitions

$$alw: \quad \square \varphi \quad \leftrightarrow \quad \varphi \wedge \bullet \square \varphi$$

Formula φ must hold now, and in the next step $\square \varphi$ again holds. Operator \square has been executed. To finish execution of the current step, sub-formula φ must also be executed. The other operators of Linear Temporal Logic can be executed similarly.

Lemma 23 (*Symbolic execution of LTL*) *The following set of rules, which are used to execute operators of Linear Temporal Logic, are sound.*

$$\begin{array}{llll} alw: & \square \varphi & \leftrightarrow & \varphi \wedge \bullet \square \varphi \\ ev: & \diamond \varphi & \leftrightarrow & \varphi \vee \circ \diamond \varphi \\ until: & \varphi \text{ until } \psi & \leftrightarrow & \psi \vee \varphi \wedge \circ (\varphi \text{ until } \psi) \\ unls: & \varphi \text{ unless } \psi & \leftrightarrow & \psi \vee \varphi \wedge \bullet (\varphi \text{ unless } \psi) \end{array}$$

Rules *alw*, *ev*, *until*, and *unls* correspond to the recursive definition of temporal operators, e.g., [10]. \square

4.12 Atomic formulas

Executing an assignment $x := e$ gives a single transition. Similar to this, atomic formulas such as PL formulas ρ , frame assumptions $[\vec{x}]$, formula **blocked**, and $\circ \varphi$ can be translated into normal form.

Lemma 24 (*Symbolic execution of atomic formulas*) *The following rules, which are used to translate atomic formulas into (weak) disjunctive normal form, are sound.*

$$\begin{array}{llll} pl: & \rho & \leftrightarrow & \rho \wedge \text{true} \wedge \text{true} \wedge \bullet \text{true} \\ frm: & [\vec{x}] & \leftrightarrow & \text{true} \wedge [\vec{x}] \wedge \text{true} \wedge \bullet \text{true} \\ blk: & \mathbf{blocked} & \leftrightarrow & \text{true} \wedge \text{true} \wedge \mathbf{blocked} \wedge \circ \text{true} \\ snx: & \circ \varphi & \leftrightarrow & \text{true} \wedge \text{true} \wedge \text{true} \wedge \circ \varphi \end{array}$$

Because of $\bullet \text{true} \leftrightarrow \text{true}$, the rules trivially hold. Note that **blocked** is equivalent to $\text{blk}' \neq \text{blk}$. Thus, the formula is not satisfied in the last state and $\bullet \text{true}$ can be strengthened to $\circ \text{true}$. \square

4.13 Propositional operators

As has been mentioned in Sect. 4.6, the propositional operators \neg , \wedge , \vee , and \rightarrow can be “symbolically executed” with the same strategy. This is demonstrated for conjunction

and implication; rules for the other logical connectives can be found in Appendix B. Conjunction is an existential operator. According to Def. 51, rules *con lem*, *con dis*, *con ex*, *con lst*, and *con stp* must be defined. However, conjunction requires both sub-formulas to be rewritten to disjunctive normal form. Therefore, rules must be defined to execute both. Rules *con lem 1*, *con lem 2*, *con dis 1*, *con dis 2* were already discussed in Section 3.3. Rules *con ex 1*, and *con ex 2* can be found in Appendix B.2. Instead of *con lst* and *con stp*, four rules are required, first in case both sub-formulas terminate (*con ll*), second and third in case the first sub-formula terminates and the second takes a step and vice versa (*con ls* and *con sl*), and fourth in case both formulas step (*con ss*).

Lemma 25 (*Conjunction of normal forms*) *The following set of rules, which are used to combine normal forms with conjunction, are sound.*

$$\begin{array}{llll}
\text{con ll:} & (\rho_1 \wedge \mathbf{last}) \wedge (\rho_2 \wedge \mathbf{last}) & \leftrightarrow & (\rho_1 \wedge \rho_2) \wedge \mathbf{last} \\
\text{con ls:} & (\rho_1 \wedge \mathbf{last}) \wedge (\tau_2 \wedge \circ \varphi_2) & \leftrightarrow & \text{false} \\
\text{con sl:} & (\tau_1 \wedge \circ \varphi_1) \wedge (\tau_2 \wedge \mathbf{last}) & \leftrightarrow & \text{false} \\
\text{con ss:} & (\rho_1 \wedge \delta_1 \wedge \beta_1 \wedge \circ \varphi_1) & \leftrightarrow & (\rho_1 \wedge \rho_2) \wedge (\delta_1 \wedge \delta_2) \\
& \wedge (\rho_2 \wedge \delta_2 \wedge \beta_2 \wedge \circ \varphi_2) & & \wedge (\beta_1 \wedge \beta_2) \wedge \circ (\varphi_1 \wedge \varphi_2)
\end{array}$$

Proving these four rules involves definitions for **last** and \circ and simple meta level reasoning. \square

The result of rule *con ss* does not adhere to normal form as the frame and block formulas are more complex. While the block formula $\beta_1 \wedge \beta_2$ can be simplified with standard propositional reasoning, additional rules are required to combine frame assumptions $\delta_1 \wedge \delta_2$.

Lemma 26 (*Conjunction of frame assumptions*) *The following rules, which are used to combine frame assumptions with conjunction, are sound.*

$$\begin{array}{llll}
\text{frm con pp:} & [\vec{x}_1] \wedge [\vec{x}_2] & \leftrightarrow & [\vec{x}_1 \cap \vec{x}_2] \\
\text{frm con pn:} & [\vec{x}_1] \wedge \neg [\vec{x}_2] & \leftrightarrow & (\bigvee_{x \in \vec{x}_1 \setminus \vec{x}_2} x' \neq x) \wedge [\vec{x}_1] \\
\text{frm con np:} & \neg [\vec{x}_1] \wedge [\vec{x}_2] & \leftrightarrow & (\bigvee_{x \in \vec{x}_2 \setminus \vec{x}_1} x' \neq x) \wedge [\vec{x}_2] \\
\text{frm con nn:} & \neg [\vec{x}_1] \wedge \neg [\vec{x}_2] & \leftrightarrow & (\bigvee_{x \in \vec{x}_1 \setminus \vec{x}_2} x' \neq x) \\
& & & \wedge (\bigvee_{x \in \vec{x}_2 \setminus \vec{x}_1} x' \neq x) \\
& & & \vee \neg [\vec{x}_1 \cup \vec{x}_2]
\end{array}$$

If the frame assumptions are interpreted as infinite conjunction of equations (see Section 2.2.7), proof of these equivalences is straightforward. \square

Simplification of frame assumptions may give additional PL formulas. The following two rules are used to rearrange the different formulas within a normal form. Rewriting with

these – admittedly technical – rules finally results in a formula which strictly adheres to normal form.

$$\begin{aligned} \text{dnf frm tau: } \rho_1 \wedge (\rho_2 \wedge \delta) \wedge \beta \wedge \varphi &\leftrightarrow (\rho_1 \wedge \rho_2) \wedge \delta \wedge \beta \wedge \varphi \\ \text{dnf frm tau: } \rho_1 \wedge \rho_2 \wedge \beta \wedge \varphi &\leftrightarrow (\rho_1 \wedge \rho_2) \wedge \text{true} \wedge \beta \wedge \varphi \end{aligned}$$

As another example, consider an implication of normal forms. Note that for an implication, the first formula must be rewritten to disjunctive, the second to conjunctive normal form. The additional rules required to combine the results are shown below.

Lemma 27 (*Implication of normal forms*) *The following set of rules, which are used to combine normal forms with implication, are sound.*

$$\begin{aligned} \text{imp ll: } \rho_1 \wedge \mathbf{last} \rightarrow \rho_2 \vee \neg \mathbf{last} &\leftrightarrow (\rho_1 \rightarrow \rho_2) \vee \neg \mathbf{last} \\ \text{imp ls: } \rho_1 \wedge \mathbf{last} \rightarrow \tau_2 \vee \bullet \varphi_2 &\leftrightarrow \text{true} \\ \text{imp sl: } \tau_1 \wedge \circ \varphi_1 \rightarrow \tau_2 \vee \neg \mathbf{last} &\leftrightarrow \text{true} \\ \text{imp ss: } \rho_1 \wedge \delta_1 \wedge \beta_1 \wedge \circ \varphi_1 &\leftrightarrow (\rho_1 \rightarrow \rho_2) \vee (\neg \delta_1 \vee \delta_2) \\ \rightarrow \rho_2 \vee \delta_2 \vee \beta_2 \vee \bullet \varphi_2 &\vee (\neg \beta_1 \vee \beta_2) \vee \bullet (\varphi_1 \rightarrow \varphi_2) \end{aligned}$$

Proving these four rules involves definitions for **last**, \circ , and \bullet . Otherwise, the proof is straightforward. \square

Again, additional rules are required to simplify $\neg \delta_1 \vee \delta_2$ and to normalize the result. The rules can be found in Appendix B.3.

4.14 Example: Executing temporal formulas

Example We prove

$$\vdash \circ \circ p \rightarrow \diamond p .$$

Informally, this property states that if p holds in the second state, then p eventually holds. Intuitively, this is sound. In order to prove this property, we translate the formula into normal form. Implication requires both sub-formulas to be rewritten first. The first formula must be transformed into (weak) disjunctive, the second into (weak) conjunctive normal form. A disjunctive normal form for the left hand side is received as follows:

$$\frac{\vdash \text{true} \wedge \text{true} \wedge \text{true} \wedge \circ \circ p \rightarrow \diamond p}{\vdash \circ \circ p \rightarrow \diamond p} \text{ snx}$$

For the right hand side, $\diamond p$ is rewritten with *ev*.

$$\frac{\vdash \text{true} \wedge \text{true} \wedge \text{true} \wedge \circ \circ p \rightarrow p \vee \circ \diamond p}{\vdash \text{true} \wedge \text{true} \wedge \text{true} \wedge \circ \circ p \rightarrow \diamond p} \text{ ev}$$

In order to rewrite the two atomic formulas p and $\circ \diamond p$ to conjunctive normal form, it is convenient to use derived rules *pl cnf* and *srx cnf* which can be found in Appendix B.3. Applied to the current goal, we receive

$$\frac{\frac{\vdash \text{true} \wedge \text{true} \wedge \text{true} \wedge \circ \circ p \rightarrow \begin{array}{c} (p \vee \text{false} \vee \text{false} \vee \circ \text{false}) \\ \vee (\text{false} \vee \text{false} \vee \text{false} \vee \circ \diamond p) \end{array}}{\vdash \text{true} \wedge \text{true} \wedge \text{true} \wedge \circ \circ p \rightarrow (p \vee \text{false} \vee \text{false} \vee \circ \text{false}) \vee \circ \diamond p} \text{ srx cnf}}{\vdash \text{true} \wedge \text{true} \wedge \text{true} \wedge \circ \circ p \rightarrow p \vee \circ \diamond p} \text{ pl cnf}$$

The disjunction of normal forms can be combined with rule *dis ww* (again, see Appendix B.3).

$$\frac{\frac{\vdash \text{true} \wedge \text{true} \wedge \text{true} \wedge \circ \circ p \rightarrow p \vee \text{false} \vee \text{false} \vee \circ \diamond p}{\vdash \text{true} \wedge \text{true} \wedge \text{true} \wedge \circ \circ p \rightarrow \begin{array}{c} (p \vee \text{false}) \vee (\text{false} \vee \text{false}) \\ \vee (\text{false} \vee \text{false}) \vee \circ (\text{false} \vee \diamond p) \end{array}} \text{ simp}}{\vdash \text{true} \wedge \text{true} \wedge \text{true} \wedge \circ \circ p \rightarrow \begin{array}{c} (p \vee \text{false} \vee \text{false} \vee \circ \text{false}) \\ \vee (\text{false} \vee \text{false} \vee \text{false} \vee \circ \diamond p) \end{array}} \text{ dis ww}$$

Finally, the formula on the right hand side of the implication is in (weak) conjunctive normal form, and the normal forms can be combined with rule *imp sw*. This rule is a variant of *imp ss* which can be used, if the right hand side is in weak conjunctive normal form.

$$\frac{\frac{\vdash p \vee \text{false} \vee \text{false} \vee \bullet (\circ p \rightarrow \diamond p)}{\vdash (\text{true} \rightarrow p) \vee (\neg \text{true} \vee \text{false}) \vee (\neg \text{true} \vee \text{false}) \vee \bullet (\circ p \rightarrow \diamond p)} \text{ simp}}{\vdash \text{true} \wedge \text{true} \wedge \text{true} \wedge \circ \circ p \rightarrow p \vee \text{false} \vee \text{false} \vee \circ \diamond p} \text{ imp sw}$$

The whole formula is in conjunctive normal form and rule *stp* can be applied.

$$\frac{\frac{\vdash \circ p \rightarrow P_0 \vee \diamond p}{\vdash P_0 \vee (\circ p \rightarrow \diamond p)} \text{ simp}}{\vdash p \vee \text{false} \vee \text{false} \vee \bullet (\circ p \rightarrow \diamond p)} \text{ stp}$$

Further first order simplification eliminates P_0 . The first step has been executed and the formula reads

$$\vdash \circ p \rightarrow \diamond p$$

As can be seen, translating propositional combinations of atomic formulas into a formula which strictly adheres to normal form is cumbersome. In the example above, it took a number of steps to rewrite a partial conjunctive normal form

$$p \vee \circ \diamond p$$

to

$$p \vee \text{false} \vee \text{false} \vee \circ \diamond p .$$

Additional rules can be derived to complete a partial normal form in a single step. In the following, completion of normal forms as well as simplification – to some extent – of the resulting premises will be implicit.

```

semaphore  $\equiv$ 
  var s = 1 in
    while true do
      (0) await s > 0; s := s - 1;
      (1) l1 := true;
      (2) {l1 := false; s := s + 1;}
      (3) skip
    ||
    while true do
      (0) await s > 0; s := s - 1;
      (1) l2 := true;
      (2) {l2 := false; s := s + 1;}
      (3) skip

```

Figure 4.3: Two processes with semaphore

Example If completion of normal forms and subsequent simplification is implicit, executing the above property reads as follows:

$$\begin{array}{c}
 \frac{\vdash \circ p \rightarrow \diamond p}{\vdash P_0 \vee (\circ p \rightarrow \diamond p)} \text{ simp} \\
 \frac{\vdash P_0 \vee (\circ p \rightarrow \diamond p)}{\vdash p \vee \bullet (\circ p \rightarrow \diamond p)} \text{ stp} \\
 \frac{\vdash p \vee \bullet (\circ p \rightarrow \diamond p)}{\vdash \circ \circ p \rightarrow p \vee \circ \diamond p} \text{ imp sw} \\
 \frac{\vdash \circ \circ p \rightarrow p \vee \circ \diamond p}{\vdash \circ \circ p \rightarrow \diamond p} \text{ ev}
 \end{array}$$

In practice, combining normal forms with propositional operators strictly simplifies a given formula. Therefore, also the application of rule *imp sw* is interpreted as a simplification and shall be implicit in the future.

Example Our proof continues with the execution of another step.

$$\begin{array}{c}
 \frac{\vdash p \rightarrow \diamond p}{\vdash P_0 \vee (p \rightarrow \diamond p)} \text{ simp} \\
 \frac{\vdash P_0 \vee (p \rightarrow \diamond p)}{\vdash p \vee \bullet (p \rightarrow \diamond p)} \text{ stp} \\
 \frac{\vdash p \vee \bullet (p \rightarrow \diamond p)}{\vdash \circ p \rightarrow \diamond p} \text{ ev}
 \end{array}$$

While executing the next step, property $\diamond p$ is finally established.

$$\frac{\frac{\vdash p \rightarrow \diamond p}{\vdash p \rightarrow p \vee \circ \diamond p} \text{ simp}}{\vdash p \rightarrow \diamond p} \text{ ev}$$

4.15 Example: Semaphore (part 1)

Example The program `semaphore` of Figure 4.3 models two processes which are repeatedly trying to access a critical section. A semaphore s is used to synchronize the access. We make

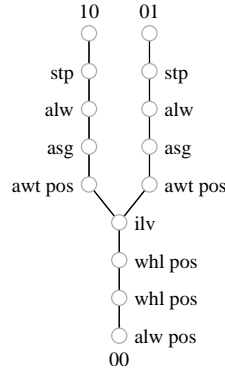


Figure 4.4: Symbolic execution of semaphore algorithm (first step)

use of two artificial boolean program variables l_1 and l_2 to signal, whether the first or the second process currently resides in its critical section. Markers $(0), (1), \dots$ are used to (informally) refer to a system configuration. The initial system configuration consists of two digits 00, where the first digit refers to the configuration of the first, the second to the configuration of the second process.

It is our task to verify that the processes do not reside in their critical sections at the same time. This property can be formalized as follows:

$$\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{semaphore} \wedge \mathbf{env} \rightarrow \mathbf{mutex}$$

where

$$\begin{aligned} \mathbf{env} &::= \square (l_1'' = l_1' \wedge l_2'' = l_2') \\ \mathbf{mutex} &::= \square (\neg l_1 \vee \neg l_2) \end{aligned}$$

Initially, labels l_1 and l_2 are false. The environment does not interfere which is ensured by formula \mathbf{env} . The mutual exclusion property is a safety property stating that at all time one of the two labels is false.

The proof strategy is to symbolically execute the given program and to verify the safety property in every step. A (partial) proof tree containing the important steps for symbolic execution can be found in Figure 4.4. Trivial proof steps, e.g., simplification have been omitted. In the beginning, the system configuration is 00. The execution of the interleaving operator is nondeterministic: either a step of the first or the second program is executed. Therefore, in the end we shall receive two premises, the system configurations being 10 and 01.

4.15.1 Executing the first step

As a rule of thumb, the operators which can be executed without case distinction should be considered first. The desired property $\neg(l_1 \wedge l_2)$ trivially holds in the first state. Therefore, rule *alw* can be applied and the additional case can be established with trivial simplification.

$$\frac{\frac{\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ (\dots \parallel \dots) \wedge \mathbf{env} \rightarrow \bullet \ \mathbf{mutex}}{\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ (\dots \parallel \dots) \wedge \mathbf{env} \rightarrow (\neg l_1 \vee \neg l_2) \wedge \bullet \ \mathbf{mutex}} \ \mathit{simp}}{\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ (\dots \parallel \dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex}} \ \mathit{alw}$$

It often occurs that a safety property $\square \varphi$ can be trivially established in the current state. An additional rule

$$\mathit{alw \ pos}: \quad \varphi \rightarrow (\square \varphi \leftrightarrow \bullet \square \varphi)$$

serves as a conditional rewrite rule. It can be automatically applied, if the precondition φ follows from the current context.

Example For each process, the condition of the **while** loop trivially holds. Similar to rule *alw pos* a derived rule *whl pos* can be applied to both processes to unwind the loop without additional cases. This rule is a variant of *whl* and can be found in the appendix. The resulting premise reads

$$\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((0)\dots \parallel (0)\dots) \wedge \mathbf{env} \rightarrow \bullet \ \mathbf{mutex} .$$

The interleaving operator is executed next. Rule *ilw* gives a disjunction of two cases. Sequential composition and local variable declarations distribute over disjunction, and therefore, we receive two premises:

$$\begin{aligned} &\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((0)\dots \parallel^< (0)\dots) \wedge \mathbf{env} \rightarrow \bullet \ \mathbf{mutex} \\ &\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((0)\dots \parallel^> (0)\dots) \wedge \mathbf{env} \rightarrow \bullet \ \mathbf{mutex} \end{aligned}$$

In both cases, the condition $s > 0$ of the await statements is trivially satisfied. For the first premise, rule *awt pos* reduces the formula representing the parallel program to

$$\mathbf{var} \ s = 1 \ \mathbf{in} \ (s := s - 1; (1)\dots) \parallel^< (0)\dots) . \quad (4.8)$$

Executing the assignment with rule *asg* and simplifying the result gives the following formula:

$$\begin{aligned} &\square \wedge \neg \mathbf{blocked} \\ &\wedge \circ \mathbf{var} \ s = 0 \ \mathbf{in} \ ((1)\dots \parallel (0)\dots) \end{aligned} \quad (4.9)$$

The assignment decrements the semaphore. Therefore, the local value of variable s will be zero in the next state. To the outside, no variable has changed. In the next state, the system continues in configuration 10.

4.15.2 Simplification

Simplifying the formula 4.8 after executing the assignment to receive formula 4.9 involves a number of steps which are given in the following. After the execution of the assignment with rule *asg*, the following formula results.

$$\begin{aligned} & \mathbf{var\ } s = 1 \mathbf{\ in} \\ & \quad (s' = s - 1 \wedge [s] \wedge \neg \mathbf{blocked} \wedge \circ \mathbf{last}); \\ & \quad (1) \dots \\ & \quad \parallel^{<} (0) \dots \end{aligned}$$

The sub-formula of the sequential composition represents a single transition in disjunctive normal form. Rewriting the composition with *chp stp* gives

$$\begin{aligned} & \mathbf{var\ } s = 1 \mathbf{\ in} \\ & \quad s' = s - 1 \wedge [s] \wedge \neg \mathbf{blocked} \wedge \circ (\mathbf{last}; (1) \dots) \\ & \quad \parallel^{<} (0) \dots \end{aligned}$$

which can be further simplified with rule *chp last* to receive

$$\begin{aligned} & \mathbf{var\ } s = 1 \mathbf{\ in} \\ & \quad s' = s - 1 \wedge [s] \wedge \neg \mathbf{blocked} \wedge \circ ((1) \dots) \\ & \quad \parallel^{<} (0) \dots \end{aligned}$$

The interleaving operator is executed next using rule *ilvl stp*.

$$\begin{aligned} & \mathbf{var\ } s = 1 \mathbf{\ in} \\ & \quad s' = s - 1 \wedge [s] \wedge \neg \mathbf{blocked} \\ & \quad \wedge \circ ((1) \dots \parallel (0) \dots) \end{aligned}$$

Finally, applying *var stp* gives

$$\begin{aligned} \exists S_0. \quad & S_0 = 1 - 1 \wedge [] \wedge \neg \mathbf{blocked} \\ & \wedge \circ \mathbf{var\ } s = S_0 \mathbf{\ in\ } ((1) \dots \parallel (0) \dots) . \end{aligned}$$

which can be simplified to

$$\begin{aligned} & [] \wedge \neg \mathbf{blocked} \\ & \wedge \circ \mathbf{var\ } s = 0 \mathbf{\ in\ } ((1) \dots \parallel (0) \dots) . \end{aligned}$$

4.15.3 Continuing with the first step

The first premise now reads

$$\begin{aligned} \vdash \quad & \neg l_1 \wedge \neg l_2 \wedge [] \wedge \neg \mathbf{blocked} \\ & \wedge \circ \mathbf{var\ } s = 0 \mathbf{\ in\ } ((1) \dots \parallel (0) \dots) \\ & \wedge \mathbf{env} \\ & \rightarrow \bullet \mathbf{mutex} . \end{aligned}$$

The program and the **mutex** property have been executed; the two formulas are preceded with a weak or strong next operator. Formula **env** remains. Rule *always* states that **env** is equivalent to

$$l_1'' = l_1' \wedge l_2'' = l_2' \wedge \bullet \mathbf{env} ,$$

i.e. the first environment transition does not modify the labels l_1 and l_2 and for the rest of the trace **env** again holds. All formulas have been executed and the reads

$$\begin{aligned} \vdash & \quad \neg l_1 \wedge \neg l_2 \wedge \square \wedge \neg \mathbf{blocked} \\ & \wedge \circ \mathbf{var} \ s = 0 \ \mathbf{in} \ ((1)\dots \parallel (0)\dots) \\ & \wedge l_1'' = l_1' \wedge l_2'' = l_2' \wedge \bullet \mathbf{env} \\ \rightarrow & \bullet \mathbf{mutex} . \end{aligned}$$

After combining the propositional combinations of normal forms, we receive

$$\begin{aligned} \vdash & \quad (\neg l_1 \wedge \neg l_2 \wedge l_1' = l_1' \wedge l_2' = l_2' \rightarrow \mathbf{false}) \\ & \vee \neg \square \vee \mathbf{blocked} \\ & \vee \bullet (\mathbf{var} \ s = 0 \ \mathbf{in} \ ((1)\dots \parallel (0)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex}) \end{aligned}$$

Rule *stp* can now be applied which first exploits the frame assumption \square to replace all primed variables with corresponding unprimed variables. Afterwards, all unprimed and primed variables are replaced by new static variables, and the double primes and leading next operator are removed. This gives

$$\begin{aligned} \vdash & \quad (\neg L_1^0 \wedge \neg L_2^0 \wedge l_1 = L_1^0 \wedge l_2'' = L_2^0 \rightarrow \mathbf{false}) \\ & \vee (\mathbf{var} \ s = 0 \ \mathbf{in} \ ((1)\dots \parallel (0)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex}) \end{aligned}$$

which can be simplified to receive

$$\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((1)\dots \parallel (0)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex} .$$

This premise is the final result of executing the first transition of the first step of program **semaphore**: the value of semaphore s is 0, and the first process has progressed. The current system configuration is 10.

Similarly, the other transition of the first step can be refined to receive

$$\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((0)\dots \parallel (1)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex} .$$

Here, the system configuration is 01; the second process has progressed.

4.15.4 Automation

Executing the first step required a number of rule applications. Figure 4.4 already gives a proof tree where all of the simplifying rules have been hidden. For example, rules to combine normal forms such as *con ss*, *var stp*, *ilw stp* etc. strictly simplify the given formula and are therefore considered simplifying rules. These rules have been annotated with (S)

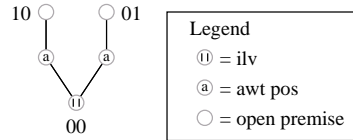


Figure 4.5: Symbolic execution of semaphore algorithm (short version of first step)

in Appendix B. Furthermore, rules to distribute operators over disjunctions or conjunctions have also been implicit. In general, there remains a single rule for each operator such as *alw*, *ilv*, *asg* with possible variations such as *alw pos*, *awt pos* to avoid additional cases. These rules can also be automatically applied such that the execution of a single step for a given proof obligation is fully automatic. For presentation purposes, we shall restrict the rules which are displayed in a proof tree to the most interesting rules. In our example, we focus on the application of *ilv* because this rule gives two cases, and *awt pos*, because the synchronization of processes is of special interest. A short version of the proof tree for executing the first step of the semaphore example is shown in Figure 4.5.

The example will be continued in the next chapter.

4.16 Conclusion

Symbolic execution is to translate a formula into normal form where the possible first transitions and the corresponding formulas describing the system in the next state are separated. Parallel programs as well as temporal formulas can be executed likewise. For translating the different operators into normal form we have defined a set of rewrite rules. The operators can be classified as follows: either the operator is directly executed or its sub-formulas are rewritten first. In the latter case, the rules adhere to a certain pattern. Following this pattern, interleaving of arbitrary temporal formulas can be executed.

We have not investigated whether the resulting set of rules is complete for every operator. However, the pattern of rules define a systematic approach to execute temporal operators for arbitrary sub-formulas. We have been able to define a set of rules according to this pattern for almost all of the operators of our logic, and so far we were able to completely execute all temporal formulas encountered in practical applications.

Symbolic execution can be fully automatic. A major number of rules strictly simplify a given formula. These rules – they are annotated with (S) in Appendix B – can be automatically applied without further consideration. While the application of a rule for symbolic execution will never invalidate a premise, the order of rule application affects the number of premises and thus proof size. As a rule of thumb, operators which do not result in additional cases should be executed first. Furthermore, additional rules can be defined to further simplify a given goal. The appendix already contains a list of additional rules for those cases which are frequently encountered. Symbolic execution works best,

if the resulting PL formulas can be automatically simplified. Thus, powerful first order simplification is required.

Chapter 5

Ingredient 2 – Sequencing

The execution of interleaved parallel processes is nondeterministic. As a consequence, a large number of different paths of execution have to be considered. Very often, however, the order of execution has no effect on the computation; following different paths leads to the same state. To minimize the number of paths, the granularity of a system step can be redefined: two statements of a process can be executed in a single step, if they do not affect the global state; if a process only assigns local variables, then the global state is not changed, and only the stuttering behavior of the process differs if statements are combined. However, this approach depends on the type of property to verify; the property must be invariant to stuttering steps.

Our approach is simple but effective. Initially, all paths of execution are considered. However, if the same system configuration results from executing a number of steps but in a different order, then the two resulting premises are combined. We do not distinguish between program and property and only, if the program configuration and the configuration of the temporal formulas are equal, the premises are combined. As a consequence, our approach is independent from the temporal property and even interleaving of abstract temporal formulas can be sequentialized.

If premises are combined, the resulting proof is a graph rather than a proof tree. An extension of the sequent calculus is presented in Section 5.1. Rules for sequencing are given in Sect. 5.2, followed by an example in Section 5.3. Section 5.4 concludes.

5.1 Proof graph

We will generalize the concept of a sequent rule to allow for several conclusions within a single rule.

$$\frac{P_1 \quad \dots \quad P_n}{C_1 \quad \dots \quad C_m} \text{ name, } \quad \text{if cond}$$

This sequent rule contains several conclusions C_1, \dots, C_m . If all of the premises P_1, \dots, P_n are valid and the side condition is satisfied, then all of the conclusions hold. A derivation – as a successive application of the more general sequent rules – is a proof graph.

Discussion: It would also be possible to define a lemma $C_1 \wedge \dots \wedge C_m$ and to verify the lemma in a separate proof

$$\frac{P_1 \quad \dots \quad P_n}{C_1 \wedge \dots \wedge C_m}$$

The lemma could then be used to prove the different conclusions of the original proof.

$$\frac{C_1 \wedge \dots \wedge C_m}{C_1} \quad \dots \quad \frac{C_1 \wedge \dots \wedge C_m}{C_m}$$

However, the possibility to verify several premises with a single proof frequently occurs in verification of interleaved parallel programs. Thus, a generalized sequent rule is more practical as it avoids the formulation of a large number of specialized lemmas with a short proof each.

The concept of proof graphs bears a resemblance to the Hilbert calculus. A derivation

$$H \equiv A_1 \dots A_n$$

in the Hilbert calculus can also be interpreted as a proof graph. However, we shall stick to the method of backward reasoning of a Sequent Calculus.

5.2 Sequencing

An appropriate rule to combine two premises is very simple.

$$\frac{\vdash \varphi \wedge \psi}{\vdash \varphi \quad \vdash \psi} \text{con}^{-1}$$

It takes two sequents $\vdash \varphi$ and $\vdash \psi$ and continues with a single premise where the conjunction of both premises is considered. It is named con^{-1} , because it is the inverse of rule

$$\frac{\vdash \varphi \quad \vdash \psi}{\vdash \varphi \wedge \psi} \text{con} .$$

Rule con^{-1} can be used to combine arbitrary premises. As a rule of thumb, two premises are combined, if the system is residing in the same configuration. A system configuration is defined by the temporal formulas of a premise. In order to automate sequencing, we define a syntactic criterion to determine the current system configuration. For this purpose, function $\text{conf}(\varphi)$ extracts a set of temporal formulas from a given formula φ .

Definition 53 (*System configuration*) The system configuration $\text{conf}(\varphi)$ of a given formula φ is a set of temporal formulas defined as follows:

$$\begin{aligned}
\text{conf}(\varphi_{\mathbf{PL}}) &::= \emptyset \\
\text{conf}(\neg \varphi) &::= \{\text{neg}(\psi) \mid \psi \in \text{conf}^-(\varphi)\} \\
\text{conf}(\varphi \vee \psi) &::= \text{conf}(\varphi) \cup \text{conf}(\psi) \\
\text{conf}(\varphi \rightarrow \psi) &::= \text{conf}(\neg \varphi) \cup \text{conf}(\psi) \\
\text{conf}(\varphi) &::= \{\varphi\} \\
\\
\text{conf}^-(\varphi_{\mathbf{PL}}) &::= \emptyset \\
\text{conf}^-(\neg \varphi) &::= \{\text{neg}\psi \mid \psi \in \text{conf}(\varphi)\} \\
\text{conf}^-(\varphi \wedge \psi) &::= \text{conf}^-(\varphi) \cup \text{conf}^-(\psi) \\
\text{conf}^-(\varphi) &::= \{\varphi\} \\
\\
\text{neg}(\neg \varphi) &::= \varphi \\
\text{neg}(\varphi) &::= \neg \varphi
\end{aligned}$$

Condition $\text{conf}(\varphi) = \text{conf}(\psi)$ is a (computable) indication to combine the two premises $\vdash \varphi$ and $\vdash \psi$. In this case, the temporal formulas of φ and ψ are equivalent. In order to simplify the combined premise, the following functions $\text{tl}(\varphi)$ and $\text{pl}(\varphi)$ are used to divide the premises into TL and PL formulas.

Definition 54 (*Dividing formulas into TL and PL formulas*) The following functions are used to divide a formula into TL and PL formulas:

$$\begin{aligned}
\text{tl}(\varphi_{\mathbf{PL}}) &::= \text{false} & \text{pl}(\varphi_{\mathbf{PL}}) &::= \varphi_{\mathbf{PL}} \\
\text{tl}(\neg \varphi) &::= \neg \text{tl}^-(\varphi) & \text{pl}(\neg \varphi) &::= \neg \text{pl}^-(\varphi) \\
\text{tl}(\varphi \rightarrow \psi) &::= \text{tl}^-(\varphi) \rightarrow \text{tl}(\psi) & \text{pl}(\varphi \rightarrow \psi) &::= \text{pl}^-(\varphi) \rightarrow \text{pl}(\psi) \\
\text{tl}(\varphi) &::= \varphi & \text{pl}(\varphi) &::= \text{false} \\
\\
\text{tl}^-(\varphi_{\mathbf{PL}}) &::= \text{true} & \text{pl}^-(\varphi_{\mathbf{PL}}) &::= \varphi_{\mathbf{PL}} \\
\text{tl}^-(\neg \varphi) &::= \neg \text{tl}(\varphi) & \text{pl}^-(\neg \varphi) &::= \neg \text{pl}(\varphi) \\
\text{tl}^-(\varphi \wedge \psi) &::= \text{tl}^-(\varphi) \wedge \text{tl}^-(\psi) & \text{pl}^-(\varphi \wedge \psi) &::= \text{pl}^-(\varphi) \wedge \text{pl}^-(\psi) \\
\text{tl}^-(\varphi) &::= \varphi & \text{pl}^-(\varphi) &::= \text{true}
\end{aligned}$$

The two functions traverse the propositional combinations and eliminate either the PL or the TL formulas. The formulas are eliminated by replacing them by false or true, depending on whether the occurrence is positive or negative. Functions conf , tl , and pl satisfy a number of properties.

Corollary 4 (*Properties of system configurations*)

1. $\models \text{tl}(\varphi) \Rightarrow \models \varphi$
2. $\models \text{pl}(\varphi) \Rightarrow \models \varphi$
3. $\models \text{pl}(\varphi) \vee \text{tl}(\varphi) \Leftrightarrow \models \varphi$
4. $\models \bigvee \text{conf}(\varphi) \Leftrightarrow \models \text{tl}(\varphi)$

For every property, we use induction over the number of leading operators $\{\neg, \rightarrow, \wedge\}$. Furthermore, the properties are generalized as follows.

1. $(\models \text{tl}(\varphi) \Rightarrow \models \varphi)$ and $(\models \varphi \Rightarrow \models \text{tl}^-(\varphi))$
2. $(\models \text{pl}(\varphi) \Rightarrow \models \varphi)$ and $(\models \varphi \Rightarrow \models \text{pl}^-(\varphi))$
3. $(\models \text{pl}(\varphi) \vee \text{tl}(\varphi) \Leftrightarrow \models \varphi)$ and $(\models \text{pl}^-(\varphi) \wedge \text{tl}^-(\varphi) \Leftrightarrow \models \varphi)$
4. $(\models \bigvee \text{conf}(\varphi) \Leftrightarrow \models \text{tl}(\varphi))$ and $(\models \bigwedge \text{conf}^-(\varphi) \Leftrightarrow \models \text{tl}^-(\varphi))$

Otherwise, the proofs are straightforward. \square

With these definitions, is it possible to define a rule more suitable for the automatic application of sequencing.

$$\frac{\vdash \text{pl}(\varphi) \wedge \text{pl}(\psi) \vee \text{tl}(\varphi)}{\vdash \varphi \quad \vdash \psi} \text{seq}, \quad \text{if } \text{conf}(\varphi) = \text{conf}(\psi)$$

Only if the system configuration of φ and ψ is equal, the premises are combined. In this case, the TL formulas of both premises are equivalent, and only the PL formulas need to be combined. Hopefully, the conjunction of PL formulas can be significantly simplified. In order to avoid complicated PL formulas it is sometimes more convenient to combine the premises only, if one of the premises is implied by the other. A stricter version seq^+ of the rule above is as follows:

$$\frac{\vdash \text{pl}(\varphi) \rightarrow \text{pl}(\psi) \quad \vdash \varphi}{\vdash \varphi \quad \vdash \psi} \text{seq}^+, \quad \text{if } \text{conf}(\varphi) \subseteq \text{conf}(\psi)$$

Here, it is sufficient that $\text{conf}(\varphi) \subseteq \text{conf}(\psi)$ as this ensures that $\text{tl}(\varphi) \rightarrow \text{tl}(\psi)$. Also, $\text{pl}(\varphi) \rightarrow \text{pl}(\psi)$, and thus, $\varphi \rightarrow \psi$.

Sequencing can also be defined on the level of single transitions:

$$\begin{aligned} \text{dis seq:} \quad & (\rho_1 \wedge \delta_1 \wedge \beta \wedge \varphi) \vee (\rho_2 \wedge \delta_2 \wedge \beta \wedge \varphi) \\ & \leftrightarrow \left(\rho_1 \wedge \bigwedge_{x \in \delta_2 \setminus \delta_1} x' = x \vee \rho_2 \wedge \bigwedge_{x \in \delta_1 \setminus \delta_2} x' = x \right) \\ & \quad \wedge \delta_1 \cup \delta_2 \wedge \beta \wedge \varphi \end{aligned}$$

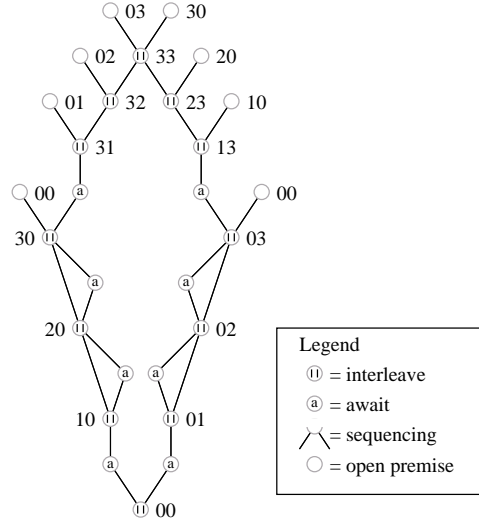


Figure 5.1: Symbolic execution of semaphore algorithm (sequencing)

If the temporal formula φ describing the system in the next state is equal in two transitions, then the transitions can be combined. In this case, it is necessary to adjust the frame assumptions δ_1 and δ_2 if they differ. This property is helpful to define an algorithm to compute the transitions of a given formula in a single step.

5.3 Example: Semaphore (part 2)

Example Consider again the example of Section 4.15. The first step of the proof obligation has been executed leaving us with two premises

$$\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((1)\dots \parallel (0)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex} . \quad (5.1)$$

and

$$\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((0)\dots \parallel (1)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex} . \quad (5.2)$$

We continue to execute the program. If symbolic execution results in the same system configuration on different branches, we apply sequencing to combine both premises. Therefore, it is best to execute the different paths of the program breadth first. A (partial) proof graph containing the important steps can be found in Figure 5.1. The proof graph only contains the most important steps.

5.3.1 Executing the second step

We now continue to execute the two premises, concentrating on the first. Again, the property $\neg l_1 \vee \neg l_2$ is trivially satisfied and *always pos* can be applied.

$$\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((1)\dots \parallel (0)\dots) \wedge \mathbf{env} \rightarrow \bullet \ \mathbf{mutex}$$

The interleaving operator of the program is executed with *ilv* leading to two premises

$$\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((1)\dots \parallel^< (0)\dots) \wedge \mathbf{env} \rightarrow \bullet \ \mathbf{mutex}$$

and

$$\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((1)\dots \parallel^> (0)\dots) \wedge \mathbf{env} \rightarrow \bullet \ \mathbf{mutex}$$

For the first premise, the assignment is executed with rule *asg* and after additional simplification, we receive

$$\begin{aligned} \vdash & \quad \neg l_1 \wedge \neg l_2 \wedge l'_1 \wedge [l_1] \wedge \neg \mathbf{blocked} \\ & \quad \wedge \circ \ \mathbf{var} \ s = 0 \ \mathbf{in} \ ((2)\dots \parallel (0)\dots) \\ & \quad \wedge \ \mathbf{env} \\ & \rightarrow \bullet \ \mathbf{mutex} \end{aligned}$$

which is reduced to

$$\vdash l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((2)\dots \parallel (0)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex} \quad (5.3)$$

after the execution of *env*, the propositional combination of normal forms, application of rule *stp* and final simplification. For the second premise, we execute the leading await statement of the second process; however, the await condition $s > 0$ is violated, and the process is blocked. Rule *awt neg* leads to

$$\begin{aligned} \vdash & \quad \neg l_1 \wedge \neg l_2 \\ & \quad \wedge \ \mathbf{var} \ s = 0 \ \mathbf{in} \ ((1)\dots \parallel^> (\top \wedge \mathbf{blocked} \wedge \circ (0)\dots)) \\ & \quad \wedge \ \mathbf{env} \\ & \rightarrow \bullet \ \mathbf{mutex} . \end{aligned}$$

which is subject to the application of rule *ilvl blk*

$$\begin{aligned} \vdash & \quad \neg l_1 \wedge \neg l_2 \\ & \quad \wedge \ \mathbf{var} \ s = 0 \ \mathbf{in} \ ((1)\dots \parallel_b^> (0)\dots) \\ & \quad \wedge \ \mathbf{env} \\ & \rightarrow \bullet \ \mathbf{mutex} . \end{aligned}$$

The process to be executed is blocked, and therefore, a step of the other process is executed simultaneously. Rule *asg* is applied to execute the assignment $l_1 := \text{true}$. Afterwards, the transitions of both processes can be combined with rule *ilvrb stp* which gives rise to the following simplified premise:

$$\begin{aligned} \vdash & \quad \neg l_1 \wedge \neg l_2 \wedge l'_1 \wedge [l_1] \wedge \neg \mathbf{blocked} \\ & \quad \wedge \circ \ \mathbf{var} \ s = 0 \ \mathbf{in} \ ((1)\dots \parallel (0)\dots) \\ & \quad \wedge \ \mathbf{env} \\ & \rightarrow \bullet \ \mathbf{mutex} \end{aligned}$$

which again is reduced to

$$\vdash l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((2)\dots \parallel (0)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex} .$$

This premise is equal to premise (5.3), and the two premises can be combined with rule *seq*. As the PL formulas in both premises are equal, the conjunction of PL formulas collapses; premise (5.3) is the single result of executing the second step of the first proof branch. The first process has taken another step now residing at configuration 2; the second process is still in configuration 0, because the step has been guarded by an await condition `await s > 0` which is currently not satisfied.

Similarly, executing premise (5.2) of the second proof branch leads to two premises which can be combined with sequencing to receive

$$\vdash \neg l_1 \wedge l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((0)\dots \parallel (2)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex} . \quad (5.4)$$

5.3.2 Executing further steps

Symbolic execution is applied over and over again, leading to the proof graph depicted in Figure 5.1. After the execution of the third step for both proof branches, our task remains to prove premises

$$\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((3)\dots \parallel (0)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex} . \quad (5.5)$$

and

$$\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((0)\dots \parallel (3)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex} . \quad (5.6)$$

Executing another step leads to four premises,

$$\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((0)\dots \parallel (0)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex} , \quad (5.7)$$

$$\vdash \neg l_1 \wedge l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((3)\dots \parallel (1)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex} , \quad (5.8)$$

$$\vdash l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((1)\dots \parallel (3)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex} , \quad (5.9)$$

and

$$\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((0)\dots \parallel (0)\dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex} . \quad (5.10)$$

The system configuration 00 in premises (5.7) and (5.10) has already been encountered, and induction is necessary to close these premises. This is explained in the next chapter. For the other premises, symbolic execution is continued for a small number of steps as can be seen in Figure 5.1 until, finally, a loop has been executed in all premises. Note that sequencing can be used to combine the two premises with configuration 33; the same configuration has been reached on two different execution paths.

5.4 Conclusion

In the example above, sequencing has been applied only sporadically. This is because the execution of the two processes is synchronized to a large extent. In other examples, sequencing is more frequently applied. Combining premises effectively avoids exponential proof size, the size of the proof being polynomial in the number of transitions of the interleaved processes. The strategy is not as effective as reducing the granularity of transitions beforehand; however, sequencing is independent from the property under verification. A more elaborate analysis of complexity remains an open issue.

Our definition of a system configuration $\text{conf}(\varphi)$ makes it possible to automatically compute a small number of candidates for sequencing. In KIV, we administrate a hash-table of premises, the hash-value being derived from the temporal formulas of a proof obligation. This makes the strategy of sequencing very efficient. Currently, premises are only combined automatically, if it can be shown with simplification alone that the PL formulas of one premise imply the formulas of the other premise (see rule seq^+). More sophisticated criteria to decide whether to combine the given PL formulas are still an open issue. In some applications it is better to consider part of the PL formulas as system description, e.g., if variables are used as program counters or state variables.

Chapter 6

Ingredient 3 – Induction

Our induction method is based on Noetherian induction. For finite intervals, it is possible to induce over the length of the trace. However, we also consider infinite intervals and therefore, in general, a well-founded ordering must be given. This is reflected in our basic induction principle of Theorem 3 in Section 6.1. In practice, the ordering can be often be derived from a known liveness property $\diamond \varphi$. In this case, we induce over the number of the first step satisfying φ (see Lem. 28 in Section 6.2). Liveness properties can be derived from other temporal operators as well. This is discussed in Sect. 6.3, where LTL operators (Sect. 6.3.1), the chop operator (Sect. 6.3.2), and interleaving (Sect. 6.3.3) are considered. The application of our induction method in practice is illustrated with an example in Section 6.4. For intuitive verification of the example, extended rules for repeated induction and for simultaneous induction are introduced.

6.1 Basic induction

Theorem 3 (Induction) Let $\mathbf{ih} := \varphi$. The following rules, which are used for induction, are sound.

$$\frac{\vdash \tau = N \wedge \bullet \square (\tau < N \rightarrow \mathbf{ih}) \rightarrow \varphi}{\vdash \varphi} \text{ind}(\tau)$$

where N is a fresh static variable $\in \mathbf{X}$ with respect to $\text{free}(\tau, \varphi)$.

$$\frac{}{\Gamma \vdash \square (\tau < N \rightarrow \mathbf{ih}) \rightarrow (\tau < N \rightarrow \text{pl}(\mathbf{ih}))} \text{ind app}, \quad \text{if } \text{conf}^-(\neg \mathbf{ih}) \subseteq \Gamma$$

$$\frac{}{\Gamma \vdash \square (\tau < N \rightarrow \mathbf{ih}) \rightarrow \bullet \square (\tau < N \rightarrow \mathbf{ih})} \text{ind weaken}, \quad \text{if } \text{conf}^-(\neg \mathbf{ih}) \not\subseteq \Gamma$$

See Appendix C.3.1. □

For a proof obligation $\vdash \varphi$, rule *ind* can be applied to receive an induction hypothesis as an additional precondition: starting with the next state, the hypothesis can always be applied, if τ has decreased, static variable N containing the original value of τ . The idea is to apply induction with rule *ind app*, if the system configuration of the induction hypothesis directly follows from the current context Γ . If this is not the case, rule *ind weaken* can be used to discard the induction hypothesis for the current state. Note that the choice between *ind app* and *ind weaken* is based on a syntactic criterion $\text{conf}^-(\neg \varphi) \subseteq \Gamma$ to ensure that the rules can be automatically applied. The two rules *ind app* and *ind weaken* are so called weakening rules (see Sect. 3.2.10).

Our strategy of symbolic execution ensures that after a finite number of steps, execution terminates or the temporal formulas loop and condition $\text{conf}^-(\neg \varphi) \subseteq \Gamma$ holds. However, it might be necessary to generalize the PL formulas with an invariant. Formulas can be generalized with weakening and strengthening rules (again see Sect. 3.2.10). It is an important principle of our induction method that only PL formulas need to be generalized.

Example Consider

$$\begin{aligned} \mathbf{INC} &::= \mathbf{while} \ n < M \ \mathbf{do} \ n := n + 1 \\ \mathbf{Env} &::= \square \ n'' = n' . \end{aligned}$$

We try to prove

$$\vdash n = 0 \wedge \mathbf{INC} \wedge \mathbf{Env} \rightarrow \diamond \mathbf{last} ,$$

i.e., the system eventually terminates. Because M is arbitrary, induction is necessary to prove this example. We generalize the PL formula $n = 0$, with an invariant $n \leq M$.

$$\frac{\vdash n = 0 \rightarrow n \leq M \quad \vdash n \leq M \wedge \mathbf{INC} \wedge \mathbf{Env} \rightarrow \diamond \mathbf{last}}{\vdash n = 0 \wedge \mathbf{INC} \wedge \mathbf{Env} \rightarrow \diamond \mathbf{last}} \textit{weaken}$$

A suitable induction term is $\tau ::= M - n$. Application of rule *ind*($M - n$) gives

$$\begin{aligned} \vdash \quad & M - n = N \wedge \bullet \square (M - n < N \rightarrow \mathbf{ih}) \\ \rightarrow & (n \leq M \wedge \mathbf{INC} \wedge \mathbf{Env} \rightarrow \diamond \mathbf{last}) \end{aligned}$$

where $\mathbf{ih} ::= n \leq M \wedge \mathbf{INC} \wedge \mathbf{Env} \rightarrow \diamond \mathbf{last}$. Formulas can be rearranged to receive

$$\begin{aligned} \vdash \quad & n \leq M \wedge M - n = N \\ & \wedge \mathbf{INC} \wedge \mathbf{Env} \\ & \wedge \bullet \square (M - n < N \rightarrow \mathbf{ih}) \\ \rightarrow & \diamond \mathbf{last} \end{aligned}$$

Proof continues with symbolic execution. If $n = M$, the while loop terminates and the property trivially holds. Thus, $n < M$ and the body of the while loop is executed. After executing the first step, we receive

$$\begin{aligned} \vdash \quad & N_0 \leq M \wedge M - N_0 = N \\ & \wedge N_0 < M \wedge N_1 = N_0 + 1 \wedge \mathbf{INC} \wedge n = N_1 \wedge \mathbf{Env} \\ & \wedge \square (M - n < N \rightarrow \mathbf{ih}) \\ \rightarrow & \diamond \mathbf{last} \end{aligned}$$

Simplification gives

$$\begin{aligned} \vdash \quad & N_0 < M \wedge n = N_0 + 1 \\ & \wedge \mathbf{INC} \wedge \mathbf{Env} \\ & \wedge \square (M - n < M - N_0 \rightarrow \mathbf{ih}) \\ \rightarrow & \diamond \mathbf{last} \end{aligned}$$

Using congruence rules to descend into the formula, we receive the following context of the induction formula:

$$\Gamma \equiv \{N_0 < M, n = N_0 + 1, \mathbf{INC}, \mathbf{Env}, \neg \diamond \mathbf{last}\} .$$

Furthermore,

$$\text{conf}^-(\neg \mathbf{ih}) \equiv \{\mathbf{INC}, \mathbf{Env}, \neg \diamond \mathbf{last}\}$$

Thus, the condition $\text{conf}^-(\neg \mathbf{ih}) \subseteq \Gamma$ is satisfied, and rule *ind app* is applied.

$$\begin{aligned} \vdash \quad & N_0 < M \wedge n = N_0 + 1 \\ & \wedge \mathbf{INC} \wedge \mathbf{Env} \\ & \wedge (M - n < M - N_0 \rightarrow (n \leq M \wedge \text{true} \wedge \text{true} \rightarrow \text{false})) \\ \rightarrow & \diamond \mathbf{last} \end{aligned}$$

where $n \leq M \wedge \text{true} \wedge \text{true} \rightarrow \text{false} \equiv \text{pl}(\mathbf{ih})$. This premise follows from two PL properties

$$\begin{aligned} \vdash N_0 < M \wedge n = N_0 + 1 &\rightarrow M - n < M - N_0 \\ \vdash N_0 < M \wedge n = N_0 + 1 &\rightarrow n \leq M \end{aligned}$$

which both can be solved by simple PL reasoning.

6.2 Induction with liveness

It is not necessary to provide a decreasing term τ in all cases. If a liveness property $\diamond \varphi$ is known, it is possible to induce over the number of steps it takes to reach the first state satisfying φ . Specialized induction rules making use of liveness properties in the antecedent can be derived from the basic induction principle of Theorem 3.

Lemma 28 (*Induction with liveness*) *Let $\mathbf{ih} \equiv \psi$. The following properties and rules are sound.*

$$\text{ev cnt: } \quad \diamond \varphi \quad \leftrightarrow \quad \exists n. n = n'' + 1 \text{ until } \varphi$$

where n is a fresh variable $\in \mathbf{Y}$ with respect to $\text{free}(\varphi)$.

$$\frac{\vdash \bullet \mathbf{ih} \text{ until } \varphi \rightarrow \psi}{\vdash \diamond \varphi \rightarrow \psi} \text{ ind ev}$$

$$\frac{}{\Gamma \vdash \mathbf{ih} \rightarrow \text{pl}(\mathbf{ih})} \text{ ind ev app, } \text{ if } \text{conf}^-(\neg \mathbf{ih}) \subseteq \Gamma$$

$$\frac{}{\Gamma \vdash \mathbf{ih} \rightarrow \text{true}} \text{ ind ev weaken, } \text{ if } \text{conf}^-(\neg \mathbf{ih}) \not\subseteq \Gamma$$

See Appendix C.3.2 □

Property *ev cnt* introduces a counter n which is decreased as long as a state satisfying φ is not yet reached. Variable n can be used as a decreasing term τ in Theorem 3. Instead of explicitly introducing variable n , a rule *ind ev* can be derived. The **until** formula of the premise ensures that **ih** can be applied as long as φ does not hold. If φ does not hold in the first state, then the **until** formula unwinds as follows.

$$\bullet \mathbf{ih} \text{ until } \varphi \quad \leftrightarrow \quad \bullet \mathbf{ih} \wedge \circ (\bullet \mathbf{ih} \text{ until } \varphi)$$

Thus, formula **ih** is available as a precondition in the next step. If the temporal formulas $\text{conf}^-(\neg \mathbf{ih})$ follow from the current context, then induction can be applied (rule *ind ev app*), otherwise **ih** is discarded for the current step (rule *ind ev weaken*).

6.3 Extracting liveness

Liveness is also contained in other temporal formulas. For example, φ **until** ψ guarantees that ψ eventually holds. Also, a safety condition $\square \varphi$ in the succedent contains liveness: the condition is equivalent to $\neg \diamond \neg \varphi$. The idea is to locate a suitable temporal formula and to turn it into an eventually property. In a first step, we show how to extract liveness from LTL operators. In a second step, we show how to exploit liveness properties, if they are part of a sequential composition or interleaving.

The idea of lifting liveness properties is as follows: consider a sequential composition

$$(\diamond \varphi_1 \wedge \varphi_2); \psi,$$

where the first program guarantees some liveness property $\diamond \varphi_1$. If φ_1 eventually holds in the first program, then φ_1 should also eventually hold for the complete program.

$$(\diamond \varphi_1 \wedge \varphi_2); \psi \quad \stackrel{?}{\leftrightarrow} \quad \diamond \varphi_1 \wedge (\varphi_2; \psi)$$

As it turns out, this property is wrong in general. However, if φ_1 is a condition (see Def. 30), the property holds.

$$(\diamond \varepsilon \wedge \varphi_2); \psi \quad \leftrightarrow \quad \diamond \varepsilon \wedge (\varphi_2; \psi)$$

Thus, reachability of a condition must be established. If a liveness property $\diamond \varphi$ is given, where φ is a temporal formula, a dynamic boolean variable l serving as a marker is introduced to establish reachability of a condition.

$$\diamond \varphi \leftrightarrow \exists l. \diamond l \wedge \neg l \text{ \textbf{until} } \varphi$$

Variable l is a fresh variable with respect to φ . The variable is such that it is false as long as φ is not satisfied.

6.3.1 LTL

Lemma 29 (*Liveness and LTL operators*) *The following properties, which are used to extract liveness from LTL operators, are sound.*

$$\begin{array}{llll} \textit{ev live:} & \diamond \varphi & \leftrightarrow & \exists l. \diamond l \wedge (\neg l) \text{ \textbf{until} } \varphi \\ \textit{until live:} & \varphi \text{ \textbf{until} } \psi & \leftrightarrow & \exists l. \diamond l \wedge (\neg l \wedge \varphi) \text{ \textbf{until} } \psi \\ \textit{alw safe:} & \square \varphi & \leftrightarrow & \forall l. \diamond l \rightarrow \varphi \text{ \textbf{unless} } l \\ \textit{unls safe:} & \varphi \text{ \textbf{unless} } \psi & \leftrightarrow & \forall l. \diamond l \rightarrow \varphi \text{ \textbf{unless} } (\psi \vee l) \end{array}$$

See Appendix C.3.3. □

As has been explained above, it is important to establish the reachability of a *condition*. The given properties introduce for the different temporal operators a variable l which acts as a trivial condition. Note that rules *alw safe* and *unls safe* are dual to *ev live* and *until live*; therefore, a negated liveness property is introduced. Informally, to prove a safety property $\square \varphi$ is to prove that φ holds on all finite prefixes of the interval. All finite prefixes are obtained by introducing a dynamic variable l which eventually holds. Property φ **unless** l requires that φ holds in all states which precede the first state satisfying l .

Example Again consider

$$\mathbf{INC} \quad \equiv \quad \mathbf{while} \ n < M \ \mathbf{do} \ n := n + 1$$

$$\mathbf{Env} \quad \equiv \quad \square n'' = n'.$$

The proof obligation is

$$\vdash n = 0 \wedge \mathbf{INC} \wedge \mathbf{Env} \rightarrow \square n \leq M$$

If n is initially zero, all runs of program **INC** always ensure that $n \leq M$ holds. The number of steps it takes to execute **INC** is M , which is a fixed but unknown value. Therefore, induction is necessary to prove that $n \leq M$ holds in all states. As an invariant, we use $n \leq M$. With rule *alw safe* a liveness property can be derived from the safety property in the succedent and

the induction rule *ind ev* can be applied. This is as follows.

$$\frac{\frac{\frac{\frac{\frac{\vdash n \leq M \wedge \mathbf{INC} \wedge \mathbf{Env} \wedge \bullet \mathbf{ih} \text{ until } l \rightarrow n \leq M \text{ unless } l}{\vdash \bullet \mathbf{ih} \text{ until } l \rightarrow (n \leq M \wedge \mathbf{INC} \wedge \mathbf{Env} \rightarrow n \leq M \text{ unless } l)}{\text{simp}}}{\vdash \diamond l \rightarrow (n = 0 \wedge \mathbf{INC} \wedge \mathbf{Env} \rightarrow n \leq M \text{ unless } l)}{\text{ind ev}}}{\vdash n = 0 \wedge \mathbf{INC} \wedge \mathbf{Env} \rightarrow \forall l. \diamond l \rightarrow n \leq M \text{ unless } l}{\text{simp}}}{\vdash n \leq M \wedge \mathbf{INC} \wedge \mathbf{Env} \rightarrow \square n \leq M}{\text{alw safe}}}{\vdash n = 0 \wedge \mathbf{INC} \wedge \mathbf{Env} \rightarrow \square n \leq M}{\text{weaken}}$$

where $\mathbf{ih} := n \leq M \wedge \mathbf{INC} \wedge \mathbf{Env} \rightarrow n \leq M \text{ unless } l$. For the premise

$$\vdash \quad \begin{array}{l} n \leq M \\ \wedge \mathbf{INC} \wedge \mathbf{Env} \\ \wedge \bullet \mathbf{ih} \text{ until } l \\ \rightarrow n \leq M \text{ unless } l \end{array}$$

a step is executed next. In this step, program \mathbf{INC} either terminates (if $\neg n < M$) or decrements n (if $n < M$). If \mathbf{INC} terminates, the property trivially holds. If \mathbf{INC} decrements n , the following premise is received.

$$\vdash \quad \begin{array}{l} N_0 \leq M \\ \wedge N_0 < M \wedge N_1 = N_0 + 1 \wedge \mathbf{INC} \wedge n = N_1 \wedge \mathbf{Env} \\ \wedge \neg L_0 \wedge \mathbf{ih} \wedge \bullet \mathbf{ih} \text{ until } l \\ \rightarrow N_0 \leq M \wedge n \leq M \text{ unless } l \end{array}$$

Simplifying the premise eliminates variables N_0 , N_1 , and L_0 and finally leads to

$$\vdash \quad \begin{array}{l} 0 < n \wedge n \leq M \\ \wedge \mathbf{INC} \wedge \mathbf{Env} \\ \wedge \mathbf{ih} \wedge \bullet \mathbf{ih} \text{ until } l \\ \rightarrow n \leq M \text{ unless } l . \end{array}$$

The temporal formulas of the context are identical to the formulas of the induction hypothesis. Therefore, rule *ind ev app* can be used to apply the hypothesis. In the resulting premise

$$\vdash \quad \begin{array}{l} 0 < n \wedge n \leq M \\ \wedge \mathbf{INC} \wedge \mathbf{Env} \\ \wedge \neg n \leq M \wedge \bullet \mathbf{ih} \text{ until } l \\ \rightarrow n \leq M \text{ unless } l , \end{array}$$

it remains to be shown that the invariant $n \leq M$ holds which is trivially the case.

6.3.2 Chop

Lemma 30 (*Liveness and chop*)

$$\text{chp live: } (\exists l. \diamond l \wedge \varphi); \psi \rightarrow \exists l_0. \diamond l_0 \wedge \varphi_0; \psi$$

where $l_0 \notin \text{free}(\psi) \cup (\text{free}(\varphi) \setminus \{l\})$ and $\varphi_0 \equiv \varphi[l_0/l]$.

See Appendix C.3.4. □

If the first program ensures reachability of a state where l holds, then the state is eventually reached for the whole system. A similar property for the second sub-program of sequential composition does not hold, as the first program is not required to terminate. Note that property *chp live* is an implication only.

Example Consider proof obligation

$$\vdash \diamond \text{last}; \diamond \text{last} \rightarrow \diamond \text{last}$$

where we assume that both parts of a sequential composition terminate and we try to prove termination of the whole system. The eventually property of the first sub-program can be used for induction as follows.

$$\frac{\frac{\frac{\frac{\frac{\vdash (\neg l \text{ until last}); \diamond \text{last} \wedge \bullet \text{ih until } l \rightarrow \diamond \text{last}}{\vdash \bullet \text{ih until } l \rightarrow ((\neg l \text{ until last}); \diamond \text{last} \rightarrow \diamond \text{last})} \text{simp}}{\vdash \diamond l \rightarrow ((\neg l \text{ until last}); \diamond \text{last} \rightarrow \diamond \text{last})} \text{ind ev}}{\vdash (\exists l. \diamond l \wedge (\neg l \text{ until last}); \diamond \text{last}) \rightarrow \diamond \text{last}} \text{simp}}{\vdash (\exists l. \diamond l \wedge \neg l \text{ until last}); \diamond \text{last} \rightarrow \diamond \text{last}} \text{chp live}}{\vdash \diamond \text{last}; \diamond \text{last} \rightarrow \diamond \text{last}} \text{ev live}}$$

where $\text{ih} \equiv (\neg l \text{ until last}); \diamond \text{last} \rightarrow \diamond \text{last}$. Proof continues with execution of a step. Either the first sub-program immediately terminates or takes a step. In the first case, we receive a premise which is trivially solved.

$$\frac{\frac{\vdash \diamond \text{last} \wedge \bullet \text{ih until } l \rightarrow \diamond \text{last}}{\vdash \text{last}; \diamond \text{last} \wedge \bullet \text{ih until } l \rightarrow \diamond \text{last}} \text{ax}}{\vdash \text{last}; \diamond \text{last} \wedge \bullet \text{ih until } l \rightarrow \diamond \text{last}} \text{simp}$$

In the second case, since the first sub-program does not terminate, l does not yet hold and the induction hypothesis is applicable in the next state. After step execution, we receive

$$\vdash (\neg l \text{ until last}); \diamond \text{last} \wedge \text{ih} \wedge \bullet \text{ih until } l \rightarrow \diamond \text{last} .$$

All temporal formulas are identical to the formulas in the induction hypothesis. Applying induction trivially closes the goal.

6.3.3 Interleaving

Lemma 31 (*Liveness and interleaving*)

$$\begin{array}{l}
\text{ilv live 1:} \quad l_1 :: (\exists l. \diamond l \wedge \varphi) \parallel l_2 :: \psi \\
\quad \rightarrow \exists l_0. \diamond l_0 \wedge (l_0 \vee l_1) :: \varphi_0 \parallel l_2 :: \psi \\
\text{ilv live 2:} \quad l_2 :: \psi \parallel l_1 :: (\exists l. \diamond l \wedge \varphi) \\
\quad \rightarrow \exists l_0. \diamond l_0 \wedge l_2 :: \psi \parallel (l_0 \vee l_1) :: \varphi_0
\end{array}$$

where $l_0 \notin \text{free}(l_1, l_2, \psi) \cup (\text{free}(\varphi) \setminus \{l\})$ and $\varphi_0 \equiv \varphi[l_0/l]$.

See Appendix C.3.5. □

If the first process ensures reachability of a state where l holds, then the state is eventually reached for the whole system. The same holds for the second process. This is true, because the semantics of interleaving is fair.

Example Consider the proof obligation

$$\vdash \diamond \text{last} \parallel \diamond \text{last} \rightarrow \diamond \text{last}$$

where two processes are interleaved for which it is only known that they terminate. In this case, also the interleaving terminates. This can be proven with induction, and the eventually property of the first process can be used instead of a termination function.

$$\frac{\frac{\frac{\vdash \diamond l \rightarrow (l :: (\neg l \text{ until last}) \parallel \diamond \text{last} \rightarrow \diamond \text{last})}{\vdash (\exists l. \diamond l \wedge (l \vee \text{false}) :: (\neg l \text{ until last}) \parallel \diamond \text{last}) \rightarrow \diamond \text{last}} \text{simp}}{\vdash (\exists l. \diamond l \wedge \neg l \text{ until last}) \parallel \diamond \text{last} \rightarrow \diamond \text{last}} \text{ilv live 1}}{\vdash \diamond \text{last} \parallel \diamond \text{last} \rightarrow \diamond \text{last}} \text{ev live}$$

The resulting liveness condition $\diamond l$ makes rule *ind ev* applicable.

$$\frac{\frac{\frac{\vdash l :: (\neg l \text{ until last}) \parallel \diamond \text{last} \wedge \bullet \text{ih until } l \rightarrow \diamond \text{last}}{\vdash \bullet \text{ih until } l \rightarrow (l :: (\neg l \text{ until last}) \parallel \diamond \text{last} \rightarrow \diamond \text{last})} \text{simp}}{\vdash \diamond l \rightarrow (l :: (\neg l \text{ until last}) \parallel \diamond \text{last} \rightarrow \diamond \text{last})} \text{ind ev}$$

where $\text{ih} \equiv l :: (\neg l \text{ until last}) \parallel \diamond \text{last} \rightarrow \diamond \text{last}$.

After induction has been initiated, a step of the sequent is executed. While rewriting the interleaving to normal form, either

- one of the programs terminates, or
- the first or the second program progresses, or
- one of the programs is blocked, and therefore both programs take a step.

This gives five cases for the interleaving.

$$\begin{aligned}
& l :: (\neg l \text{ until last}) \parallel \diamond \text{ last} \\
\leftrightarrow & \diamond \text{ last} \\
& \vee \neg l \text{ until last} \\
& \vee \neg l \wedge \circ (l :: (\neg l \text{ until last}) \parallel \diamond \text{ last}) \\
& \vee \neg l \wedge \circ (l :: (\neg l \text{ until last}) \parallel \diamond \text{ last}) \\
& \vee \neg l \wedge \circ (l :: (\neg l \text{ until last}) \parallel \diamond \text{ last})
\end{aligned}$$

The first case is trivial, and the last three cases are identical. Therefore, only two premises remain.

$$\begin{array}{c}
(1) \vdash \neg l \text{ until last} \wedge \bullet \text{ ih until } l \rightarrow \diamond \text{ last} \\
(2) \vdash \neg l \wedge \circ (l :: (\neg l \text{ until last}) \parallel \diamond \text{ last}) \wedge \bullet \text{ ih until } l \\
\quad \rightarrow \diamond \text{ last} \\
\hline
\vdash l :: (\neg l \text{ until last}) \parallel \diamond \text{ last} \wedge \bullet \text{ ih until } l \rightarrow \diamond \text{ last} \quad \textit{exec}
\end{array}$$

In the first premise (1), the program has permanently changed. Therefore, the induction hypothesis cannot be applied anymore and can be discarded. Discarding induction involves eliminating variable l .

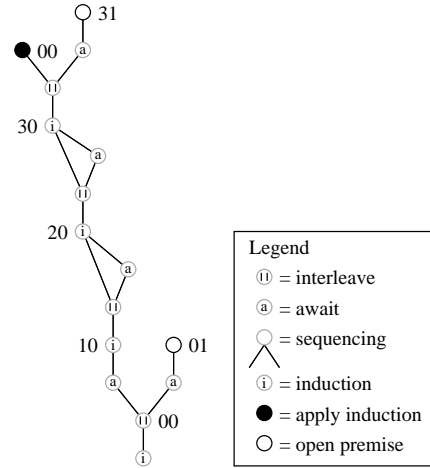
$$\begin{array}{c}
\frac{\frac{\frac{\vdash \diamond \text{ last} \rightarrow \diamond \text{ last}}{ax}}{\vdash \text{ true until last} \wedge \text{ true} \rightarrow \diamond \text{ last}} \textit{simp}}{\vdash \neg l \text{ until last} \wedge \bullet \text{ ih until } l \rightarrow \diamond \text{ last}} \textit{weaken}
\end{array}$$

After weakening formulas involving l , the premise can be simplified. Especially, true until last can be rewritten to $\diamond \text{ last}$. The remaining premise is trivial.

For the second premise (2), executing a step is completed.

$$\begin{array}{c}
\frac{\frac{\frac{\vdash l :: (\neg l \text{ until last}) \parallel \diamond \text{ last} \wedge \bullet \text{ ih until } l \rightarrow \diamond \text{ last}}{\textit{ind ev app}}}{\vdash \neg l \wedge \circ (l :: (\neg l \text{ until last}) \parallel \diamond \text{ last}) \wedge \bullet \text{ ih until } l \rightarrow \diamond \text{ last}} \textit{simp}}{\vdash \neg l \wedge \circ (l :: (\neg l \text{ until last}) \parallel \diamond \text{ last}) \wedge \bullet \text{ ih until } l \rightarrow \diamond \text{ last}} \textit{step}
\end{array}$$

The temporal formulas of the resulting premise are identical to the formulas in the induction hypothesis and induction can be applied with rule *ind ev app*. As the invariant is true, the proof is finished.

Figure 6.1: Induction for property **mutex** of semaphore algorithm

6.4 Example: Semaphore (part 3)

Example Consider again the example of Sect. 5.3 where a simple parallel program with two processes accessing a critical section has been examined. We have shown how to symbolically execute the parallel program and how to apply sequencing to reduce the number of paths to be considered. The two processes never terminate and therefore, induction is necessary to complete the proof. The idea is to execute the program until a program configuration is reached which has already been encountered. In this case, we either apply sequencing, if the identical program configuration can be found on the same level in the proof but on a different branch, or we apply induction, if it occurred in an earlier step. This is described next.

A proof graph containing the important proof steps of the first part of the proof can be found in Figure 6.1. Trivial proof rules have been omitted.

6.4.1 Induction

Both processes in the initial proof obligation loop infinitely often. Therefore, symbolic execution alone does not terminate and induction must be applied. For technical reasons, we first unwind the while loops of both processes using rule *while loop*.

$$\vdash \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((\theta) \dots \parallel (\theta) \dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex}$$

The property to verify is a safety property $\square \neg l_1 \vee \neg l_2$ which corresponds to a negated liveness property $\diamond l_1 \wedge l_2$ in the antecedent. Informally, we assume that there is a state which violates the safety property $\neg l_1 \vee \neg l_2$, and we induce over the number of steps it

takes to reach this state. Technically, a liveness property $\diamond l$ can be derived from the safety property in the succedent with rule *alw safe*.

$$\frac{\begin{array}{l} \vdash \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((0) \dots \parallel (0) \dots) \wedge \mathbf{env} \\ \rightarrow \forall l. \diamond l \rightarrow \mathbf{mutex}_l \end{array}}{\vdash \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((0) \dots \parallel (0) \dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex}} \textit{alw safe}$$

where $\mathbf{mutex}_l \equiv (\neg l_1 \vee \neg l_2) \ \mathbf{unless} \ l$. Simplifying the premise results in

$$\frac{\begin{array}{l} \vdash \quad \diamond l \\ \rightarrow \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((0) \dots \parallel (0) \dots) \wedge \mathbf{env} \\ \rightarrow \mathbf{mutex}_l \end{array}}{\vdash \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((0) \dots \parallel (0) \dots) \wedge \mathbf{env} \rightarrow \forall l. \diamond l \rightarrow \mathbf{mutex}_l} \textit{simp}$$

With the derived liveness property, rule *ind ev* is applicable. It is not necessary to generalize the PL formulas as after executing the while loops of the two parallel processes, the state of the labels L_i , M_j and of the semaphore S is again the same. Applying *ind ev* and simplifying the result gives

$$\begin{array}{l} \vdash \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((0) \dots \parallel (0) \dots) \wedge \mathbf{env} \\ \wedge \bullet \mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \ \mathbf{until} \ l \\ \rightarrow \mathbf{mutex}_l, \end{array} \quad (6.1)$$

where

$$\mathbf{ih}_{l,m}^{\varphi_1, \varphi_2, s} \quad \equiv \quad \varphi_1 \wedge \varphi_2 \wedge \mathbf{var} \ s = s \ \mathbf{in} \ ((l): \dots \parallel (m): \dots) \wedge \mathbf{env} \rightarrow \mathbf{mutex}_l$$

6.4.2 Executing the first step

As has been explained in Sect. 4.15, symbolic execution of the first step gives two premises

$$\begin{array}{l} \vdash \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((1) \dots \parallel (0) \dots) \wedge \mathbf{env} \\ \wedge \mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \wedge \bullet \mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \ \mathbf{until} \ l \\ \rightarrow \mathbf{mutex}_l, \end{array}$$

and

$$\begin{array}{l} \vdash \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((0) \dots \parallel (1) \dots) \wedge \mathbf{env} \\ \wedge \mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \wedge \bullet \mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \ \mathbf{until} \ l \\ \rightarrow \mathbf{mutex}_l. \end{array}$$

In both premises, the induction hypothesis is not yet applicable, and therefore, $\mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1}$ can be discarded leaving us with

$$\begin{array}{l} \vdash \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((1) \dots \parallel (0) \dots) \wedge \mathbf{env} \\ \wedge \bullet \mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \ \mathbf{until} \ l \\ \rightarrow \mathbf{mutex}_l, \end{array} \quad (6.2)$$

$$\begin{aligned} \vdash \quad & \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((0) \ \dots \ || \ (1) \ \dots) \wedge \mathbf{env} \\ & \wedge \bullet \mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \ \mathbf{until} \ l \\ \rightarrow & \mathbf{mutex}_l . \end{aligned} \quad (6.3)$$

These two premises are the final result of executing the two transitions which are possible in the first state: in premise (6.2), the first process has been executed and the system is in configuration 10, in the second premise (6.3), the second process has been executed and the configuration is 01. In both cases, the value of semaphore s is 0. We continue our proof with the first premise.

6.4.3 Repeated induction

The first induction alone may not be sufficient to complete the proof as the two interleaved processes can be executed such that configuration 00 will never hold again. As an example, consider a system run where the second process completes a loop returning to configuration 0 only if the first process resides at configuration 1 at the same time. Thus, instead of 00, the program configuration 10 is repeatedly encountered. In order to complete the proof in these cases, the current premise is added to the induction hypothesis with rule *rep ind*.

Lemma 32 (*Repeated induction*) *Let $\mathbf{ih} \equiv \psi$. The following rule is sound.*

$$\frac{\vdash \bullet (\mathbf{ih}_0 \wedge \mathbf{ih}) \ \mathbf{until} \ \varphi \rightarrow \psi}{\vdash \bullet \mathbf{ih}_0 \ \mathbf{until} \ \varphi \rightarrow \psi} \ \mathit{rep \ ind}$$

See Appendix C.3.6. □

Example Applied to (6.2), we receive

$$\begin{aligned} \vdash \quad & \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((1) \ \dots \ || \ (0) \ \dots) \wedge \mathbf{env} \\ & \wedge \bullet (\mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \wedge \mathbf{ih}_{1,0}^{\neg l_1, \neg l_2, 0}) \ \mathbf{until} \ l \\ \rightarrow & \mathbf{mutex}_l , \end{aligned}$$

With the two resulting induction formulas $\mathbf{ih}_{\neg l_1, \neg l_2, 1, 0, 0}$ and $\mathbf{ih}_{\neg l_1, \neg l_2, 0, 1, 0}$ induction can be applied, if configuration 00 and 10 again holds.

As a rule of thumb, the premise is added to the induction, if one process is at the beginning of a loop, while the other processes reside somewhere within a loop.

6.4.4 Executing further steps

Example In premise (6.2), the first process has entered its critical section and the second

process is blocked. Therefore, executing the second step gives a single premise (see Sect. 5.3)

$$\begin{aligned} \vdash & \quad l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((2) \dots \parallel (0) \dots) \wedge \mathbf{env} \\ & \wedge \mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \wedge \mathbf{ih}_{1,0}^{\neg l_1, \neg l_2, 0} \\ & \wedge \bullet (\mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \wedge \mathbf{ih}_{1,0}^{\neg l_1, \neg l_2, 0}) \ \mathbf{until} \ l \\ & \rightarrow \mathbf{mutex}_l, \end{aligned}$$

None of the two induction hypothesis can be applied. Therefore, they are discarded leaving us with

$$\begin{aligned} \vdash & \quad l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((2) \dots \parallel (0) \dots) \wedge \mathbf{env} \\ & \wedge \bullet (\mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \wedge \mathbf{ih}_{1,0}^{\neg l_1, \neg l_2, 0}) \ \mathbf{until} \ l \\ & \rightarrow \mathbf{mutex}_l, \end{aligned}$$

The strategy of repeated induction and step execution is continued. After the execution of the third step, our task remains to prove

$$\begin{aligned} \vdash & \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((3) \dots \parallel (0) \dots) \wedge \mathbf{env} \\ & \wedge \bullet (\mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \wedge \mathbf{ih}_{1,0}^{\neg l_1, \neg l_2, 0} \wedge \mathbf{ih}_{2,0}^{l_1, \neg l_2, 0}) \ \mathbf{until} \ l \\ & \rightarrow \mathbf{mutex}_l, \end{aligned}$$

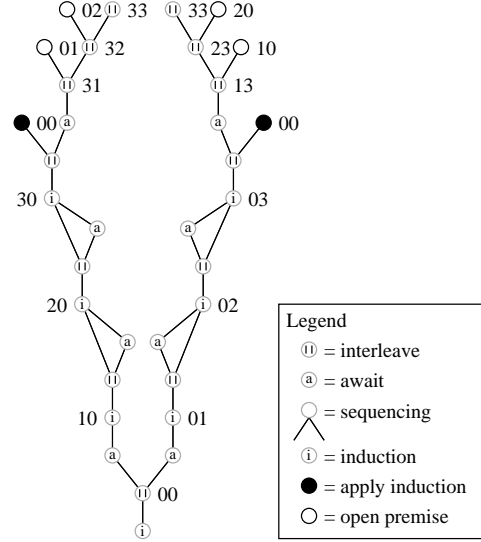
Note that $\mathbf{ih}_{2,0}^{l_1, \neg l_2, 0}$ has been added to the sequent. Executing another step leads to two premises,

$$\begin{aligned} \vdash & \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((0) \dots \parallel (0) \dots) \wedge \mathbf{env} & (6.4) \\ & \wedge \mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \\ & \wedge \bullet \quad \left(\mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \wedge \mathbf{ih}_{1,0}^{\neg l_1, \neg l_2, 0} \wedge \mathbf{ih}_{2,0}^{l_1, \neg l_2, 0} \right. \\ & \quad \left. \wedge \mathbf{ih}_{3,0}^{\neg l_1, \neg l_2, 1} \right) \\ & \quad \mathbf{until} \ l \\ & \rightarrow \mathbf{mutex}_l, \end{aligned}$$

and

$$\begin{aligned} \vdash & \quad \neg l_1 \wedge l_2 \wedge \mathbf{var} \ s = 0 \ \mathbf{in} \ ((3) \dots \parallel (1) \dots) \wedge \mathbf{env} & (6.5) \\ & \wedge \bullet \quad \left(\mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \wedge \mathbf{ih}_{1,0}^{\neg l_1, \neg l_2, 0} \wedge \mathbf{ih}_{2,0}^{l_1, \neg l_2, 0} \right. \\ & \quad \left. \wedge \mathbf{ih}_{3,0}^{\neg l_1, \neg l_2, 1} \right) \\ & \quad \mathbf{until} \ l \\ & \rightarrow \mathbf{mutex}_l, \end{aligned}$$

In the first premise, a configuration 00 is encountered which is syntactically equal to the configuration of $\mathbf{ih}_{\neg l_1, \neg l_2, 1, 0, 0}$. Therefore, the induction hypothesis has not been discarded; instead, the hypothesis can be applied.

Figure 6.2: Proof of property **mutex** of semaphore algorithm

Applying induction

For premise (6.4), every temporal formula is syntactically equal to a temporal formula in the induction hypothesis $\mathbf{ih}_{1,0,0}$. Therefore, rule *ind ev app* can be applied.

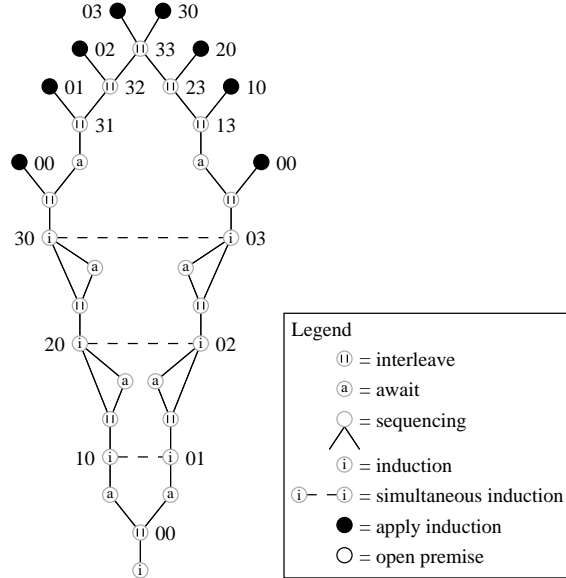
$$\begin{array}{l}
 \vdash \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((0) \dots \parallel (0) \dots) \wedge \mathbf{env} \\
 \quad \wedge \neg (\neg l_1 \wedge \neg l_2) \\
 \quad \wedge \dots \\
 \rightarrow \mathbf{mutex}_l, \\
 \hline
 \vdash \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var} \ s = 1 \ \mathbf{in} \ ((0) \dots \parallel (0) \dots) \wedge \mathbf{env} \quad \textit{ind ev app} \\
 \quad \wedge \mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \\
 \quad \wedge \dots \\
 \rightarrow \mathbf{mutex}_l,
 \end{array}$$

The invariant trivially holds as the labels $\neg l_1 \wedge \neg l_2$ is again satisfied.

This finishes the proof for the system run where only the first process steps, while the second process idles. How to verify the system runs where steps of the two processes are interleaved is discussed next.

6.4.5 Simultaneous induction

Continuing with the proof of Fig. 6.1, the open premises which are labeled 31 and 01 must be considered. If steps are executed until a program configuration is reached, which

Figure 6.3: Final Proof of property **mutex** of semaphore algorithm

already occurred, the proof tree of Fig. 6.2 results. To finish the proof, two kinds of problems remain which are both related to induction:

- Configuration 01 in the left part of the proof has already been encountered in the right part. However, the induction hypothesis $\mathbf{ih}_{\neg 1_1, 1_2, 0, 0, 1}$ is not part of the premise and induction cannot be applied.
- The two premises with configuration 33 cannot be united with sequencing as the induction formula differs.

Our solution to both is the concept of simultaneous induction. The idea is – earlier in the proof – to combine the two goals 10 and 01 with conjunction and to apply induction to the combined goal. The goals can again be separated with the effect that both induction hypothesis are part of both premises!

A rule for simultaneous induction is introduced with the following lemma, the corresponding proof illustrates the underlying idea.

Lemma 33 (*Simultaneous induction*)

$$\frac{\begin{array}{l} \vdash \bullet (\mathbf{ih}_0 \wedge \mathbf{ih}_1 \wedge \mathbf{ih}_2) \text{ until } \varphi \vdash \psi_1 \\ \vdash \bullet (\mathbf{ih}_0 \wedge \mathbf{ih}_1 \wedge \mathbf{ih}_2) \text{ until } \varphi \vdash \psi_2 \end{array}}{\vdash \bullet \mathbf{ih}_0 \text{ until } \varphi \rightarrow \psi_1 \quad \vdash \bullet \mathbf{ih}_0 \text{ until } \varphi \rightarrow \psi_2} \text{sim ind}$$

where $\mathbf{ih}_1 := \psi_1$ and $\mathbf{ih}_2 := \psi_2$.

For the two premises, sequencing is applied. Afterwards rule *rep ind* is used and the resulting premise are again separated.

$$\begin{array}{c}
 \frac{\frac{\frac{\frac{\vdash \bullet (\mathbf{ih}_0 \wedge \mathbf{ih}_1 \wedge \mathbf{ih}_2) \text{ until } \varphi \vdash \psi_1}{\vdash \bullet (\mathbf{ih}_0 \wedge \mathbf{ih}_1 \wedge \mathbf{ih}_2) \text{ until } \varphi \vdash \psi_2}}{\vdash \bullet (\mathbf{ih}_0 \wedge \mathbf{ih}_1 \wedge \mathbf{ih}_2) \text{ until } \varphi \rightarrow \psi_1 \wedge \psi_2} \text{ con } r}{\vdash \bullet (\mathbf{ih}_0 \wedge \mathbf{ih}_1 \wedge \mathbf{ih}_2) \text{ until } \varphi \rightarrow \psi_1 \wedge \psi_2} \text{ rep ind}}{\vdash (\bullet \mathbf{ih}_0 \text{ until } \varphi \rightarrow \psi_1) \wedge (\bullet \mathbf{ih}_0 \text{ until } \varphi \rightarrow \psi_2)} \text{ pl}}{\vdash \bullet \mathbf{ih}_0 \text{ until } \varphi \rightarrow \psi_1 \quad \vdash \bullet \mathbf{ih}_0 \text{ until } \varphi \rightarrow \psi_2} \text{ con}^{-1}
 \end{array}$$

□

Example Applied to premises (6.2) and (6.3), we receive

$$\begin{array}{c}
 \vdash \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var } s = 0 \text{ in } ((1) \dots \parallel (0) \dots) \wedge \mathbf{env} \\
 \wedge \bullet (\mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \wedge \mathbf{ih}_{1,0}^{\neg l_1, \neg l_2, 0} \wedge \mathbf{ih}_{0,1}^{\neg l_1, \neg l_2, 0}) \text{ until } l \\
 \rightarrow \mathbf{mutex}_l \\
 \vdash \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var } s = 0 \text{ in } ((0) \dots \parallel (1) \dots) \wedge \mathbf{env} \\
 \wedge \bullet (\mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \wedge \mathbf{ih}_{1,0}^{\neg l_1, \neg l_2, 0} \wedge \mathbf{ih}_{0,1}^{\neg l_1, \neg l_2, 0}) \text{ until } l \\
 \rightarrow \mathbf{mutex}_l \\
 \hline
 \vdash \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var } s = 0 \text{ in } ((1) \dots \parallel (0) \dots) \wedge \mathbf{env} \\
 \wedge \bullet \mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \text{ until } l \\
 \rightarrow \mathbf{mutex}_l \\
 \vdash \quad \neg l_1 \wedge \neg l_2 \wedge \mathbf{var } s = 0 \text{ in } ((0) \dots \parallel (1) \dots) \wedge \mathbf{env} \\
 \wedge \bullet \mathbf{ih}_{0,0}^{\neg l_1, \neg l_2, 1} \text{ until } l \\
 \rightarrow \mathbf{mutex}_l
 \end{array} \text{ sim ind}$$

The induction hypothesis $\mathbf{ih}_{0,1}^{\dots}$ now is available on the left branch of the proof.

If simultaneous induction is consequently applied on every proof level, the proof of Fig 6.3 is received. For premise 01 induction can be applied, because $\mathbf{ih}_{0,1}^{\dots}$ is part of the left proof branch. Furthermore, the two proof branches can be sequentialized in configuration 33, because the induction formula is equal in both proof branches.

6.4.6 Summary

Iterated application of symbolic execution with induction and sequencing is sufficient to prove the safety property **mutex**. For symbolic execution, it is either obvious which rule to apply next or the order of application does not matter. Therefore, the system can be executed fully automatic. For the example, induction does not require an invariant and can therefore be automatically initiated as well. Finally, the criteria to apply sequencing or to apply an induction hypothesis is purely syntactic – the corresponding temporal formulas are syntactically equal – which makes the complete proof for the safety property **mutex** fully automatic.

6.5 Automation

We have shown how to use induction to execute infinite system runs: the PL formulas of a sequent is generalized with an invariant; either an induction term or a liveness condition is used to construct an induction hypothesis which contains the invariant and the temporal formulas of the current sequent. The sequent is executed until a premise is reached where the system configuration (the temporal formulas) are syntactically equal to the system configuration in the induction hypothesis. In this case, the hypothesis is applied; it remains to be shown that the induction term has decreased and that the invariant again holds.

In practice, it is necessary to repeatedly use (simultaneous) induction. As a rule of thumb, induction is used, if one of the interleaved processes is at the beginning of a loop, while the other processes reside somewhere within a loop.

While – in general – the invariant and the induction term cannot be automatically found, the extraction of known liveness properties is automatable. Furthermore, it can be automatically decided whether induction is necessary or not and when to apply an induction hypothesis, because both decisions are based on syntactic criteria.

6.6 Completeness

Execution of a program or temporal formula either terminates or leads to a premise, where the temporal formulas are syntactically equal to the temporal formulas in a premise earlier in the proof. *Therefore, induction can always be applied after a finite number of steps.* However, a suitable invariant must be found and thus, the completeness of the approach is relative to the underlying domain: the invariant must be expressible with the specified algebraic operations.

Another issue is the derivation of liveness. If the sequent contains liveness, it is not necessary to supply an induction term. We have given a number of rules to extract liveness from certain temporal formulas and have also shown how to lift liveness properties which are part of a sequential or parallel program. So far, we have not defined rules for every temporal operator or construct of a parallel program. It remains an open issue to complete the set of rules and to study whether a complete set of rules can be found.

6.7 Summary

Independent from the kind of temporal property to verify, our induction is always based on the same induction principle and the proof method is always the same: use (simultaneous) induction to add the current system configuration to the induction hypothesis and execute the system until a system configuration is again encountered. This strategy can be used

to verify safety and liveness properties of sequential or parallel programs, and even to establish tautologies in pure temporal logic.

In our approach it is only necessary to generalize PL formulas. Furthermore, the invariant need not hold in every state along the trace. It is only required to establish a loop for which the invariant holds before and after the complete execution of this loop. This is different from [20] where a formula must be found which is invariant to every transition. In comparison, our invariants are shorter and should therefore be easier to find.

It can be automatically decided when to use induction and when to apply the induction hypothesis. Furthermore, liveness properties can be automatically extracted. It remains to manually provide invariants and to come up with an induction term, if the current premise does not contain liveness.

Chapter 7

Ingredient 4 – Abstraction

Our approach to decompose proofs is to abstract complex systems by temporal properties. In principle, a proof obligation

$$\vdash \varphi_1 \parallel \psi_1 \rightarrow \chi$$

can be decomposed into three proofs

1. $\vdash \varphi_1 \rightarrow \varphi_2$
2. $\vdash \psi_1 \rightarrow \psi_2$
3. $\vdash \varphi_2 \parallel \psi_2 \rightarrow \chi$

In practice, (simple) abstract temporal properties φ_2 and ψ_2 are easier to execute, because they loop after a smaller number of steps compared to the original systems φ_1 and ψ_1 .

To replace an interleaved process φ_1 by an abstract property φ_2 is sound because our semantics of interleaving is compositional. This is explained in Section 7.1. Not only interleaving, but all of our operators are compositional. Even more, arbitrary operators with an operational semantics in the style of Section 2.5 can be decomposed (see Thm. 4 in Section 7.2). An example in Section 7.3 illustrates how abstraction results in shorter proofs. The abstraction in our example is closely related to the notion of verification diagrams in STeP. A comparison can be found in Sect. 7.4. Section 7.5 concludes.

7.1 Congruence rules

Abstraction is manifested in so called congruence rules. Interleaving, for example, satisfies a property

$$\text{inv lem 1: } [\text{true}] \bullet (\varphi_1 \rightarrow \varphi_2) \rightarrow (l_1 :: \varphi_1 \parallel l_2 :: \psi \rightarrow l_1 :: \varphi_2 \parallel l_2 :: \psi)$$

for which the following congruence rule can be derived:

$$\frac{\Gamma_0 \vdash_{\mathbf{c}} \varphi_1 \subset \varphi_2 \quad \Gamma \vdash_{\mathbf{c}} l_1 :: \varphi_2 \parallel l_2 :: \psi \subset \chi}{\Gamma \vdash_{\mathbf{c}} l_1 :: \varphi_1 \parallel l_2 :: \psi \subset \chi} \text{ } ilv \text{ } lem \text{ } 1^{\mathbf{c}}$$

where $\Gamma_0 := \{\varepsilon_{\mathbf{s}} \mid \varepsilon_{\mathbf{s}} \in \Gamma, \varepsilon_{\mathbf{s}} \text{ is a static expression}\}$ (see Appendix A). The context modifier $[\text{true}] \bullet (\varphi_1 \rightarrow \varphi_2)$ of property *ilv lem 1* ensures that property $\varphi_1 \rightarrow \varphi_2$ holds for all paths and for all initial states. In the resulting congruence rule, only static context Γ_0 can be used to establish $\varphi_1 \subset \varphi_2$. In practice, the congruence rule is applied as follows:

$$\frac{\vdash \varphi_1 \rightarrow \varphi_2 \quad \vdash l_1 :: \varphi_2 \parallel l_2 :: \psi \rightarrow \chi}{\vdash l_1 :: \varphi_1 \parallel l_2 :: \psi \rightarrow \chi} \text{ } weaken$$

In order to verify a property χ for two interleaved processes φ_1 and ψ , the process φ_1 can be replaced by an abstract property φ_2 , if it can be independently established that the process implies the property. The replacement is sound, because – due to the introduction of double primed variables – every possible environment is considered in the semantics. To prove $\vdash \varphi_1 \rightarrow \varphi_2$ is to verify $\varphi_1 \rightarrow \varphi_2$ for all environments.

The semantics in Section 2 is such that congruence rules can be derived for all operators. In other words, all of the operators are compositional and sub-formulas can be abstracted.

7.2 Compositional operational semantics

For all operators where an operational semantics using SOS rules is given in the style of Sect. 2.5, a congruence rule directly holds independent from the definition of the transition relation. This property is captured in the following theorem.

Theorem 4 (*Compositional operational semantics*) Consider an operator \oplus with sub-formula φ . If the semantics of the operator is operational and is defined as follows

$$I \models \oplus(\varphi) \quad \text{iff} \quad \begin{array}{l} \text{there exist } I_0 \\ \text{with } I \in \llbracket \oplus(I_0) \rrbracket \text{ and } I_0 \models \varphi \end{array}$$

with a function

$$\llbracket \oplus(\cdot) \rrbracket : \mathbf{I} \mapsto \mathcal{P}^{\mathbf{I}}$$

defining the possible intervals which result, if the operator is applied to a concrete interval I , then the operator is compositional and the following property holds.

$$\oplus \text{ } lem: \quad [\text{true}] \bullet (\varphi_1 \rightarrow \varphi_2) \rightarrow (\oplus(\varphi_1) \rightarrow \oplus(\varphi_2))$$

The semantics of interleaving (see Sect. 2.6) and nondeterministic choice (see Sect. 2.7) has been defined in the required style. Thus, the operators are compositional independent

from the actual SOS rules. The simple reason for this surprising property can be illustrated for interleaving: the semantics of $\varphi \parallel \psi$ is reduced to the interleaving of intervals $I_1 \parallel I_2$ where I_1 and I_2 are concrete runs of φ and ψ .

(Proof for Theorem 4) The overall proof is very simple. Property $\oplus lem$ can be expanded as follows:

$$\begin{aligned} & I \models [\text{true}] \bullet (\varphi_1 \rightarrow \varphi_2) \rightarrow (\oplus(\varphi_1) \rightarrow \oplus(\varphi_1)) \\ \Leftrightarrow & I \models [\text{true}] \bullet (\varphi_1 \rightarrow \varphi_2) \text{ and } I \models \oplus(\varphi_1) \Rightarrow I \models \oplus(\varphi_1) \quad \text{Sem. } \rightarrow \end{aligned}$$

With the first precondition, we receive

$$\begin{aligned} & I \models [\text{true}] \bullet (\varphi_1 \rightarrow \varphi_2) \\ \Leftrightarrow & I_0 \models \varphi_1 \rightarrow \varphi_2 \text{ for all } I_0 \quad \text{Cor. 1} \\ \Leftrightarrow & I_0 \models \varphi_1 \Rightarrow I \models \varphi_2 \text{ for all } I_0 \quad \text{Def. } \rightarrow \end{aligned}$$

Property $I \models \oplus(\varphi_2)$ can then be easily derived from the second precondition.

$$\begin{aligned} & I \models \oplus(\varphi_1) \\ \Leftrightarrow & \text{there exists } I_0 \text{ with } I \in \llbracket \oplus(I_0) \rrbracket \text{ and } I_0 \models \varphi_1 \quad \text{Sem. } \oplus \\ \Rightarrow & \text{there exists } I_0 \text{ with } I \in \llbracket \oplus(I_0) \rrbracket \text{ and } I_0 \models \varphi_2 \quad \text{First precondition.} \\ \Leftrightarrow & I \models \oplus(\varphi_2) \quad \text{Sem. } \oplus \end{aligned}$$

□

Furthermore, the operator is an existential operator (according to the classification of operators in Sect. 4.6) and rules $\oplus dis$ and $\oplus ex$ automatically hold. Rules $\oplus lst$ and $\oplus stp$ remain to be defined.

Lemma 34 *If an operator \oplus is defined as above, the following rules hold*

$$\begin{aligned} \oplus dis: & \quad \oplus(\varphi_1 \vee \varphi_2) \quad \leftrightarrow \quad \oplus(\varphi_1) \vee \oplus(\varphi_2) \\ \oplus ex: & \quad \oplus(\exists x. \varphi) \quad \leftrightarrow \quad \exists x. \oplus(\varphi) \end{aligned}$$

Proofs are similar to the proof of Theorem 4. □

Further interesting operators such as synchronous interleaving, interrupts, etc. could be defined with appropriate SOS rules. According to Theorem 4, all of the operators would be compositional.

7.3 Example: Handshaking

Example Consider again the example of Section 1.5.6.

$$\vdash \mathbf{P}_1 \parallel \mathbf{P}_2 \wedge \mathbf{Env} \rightarrow (0) \square (l'_1 \neq l_1 \wedge D = i \rightarrow (1) \diamond (l'_2 \neq l_2 \wedge o = D))$$

where

$$\begin{aligned}
\mathbf{P}_1 &::= \mathbf{while\ true\ do} \\
&\quad (0) \quad i := ?; \\
&\quad (1) \quad l_1 : \mathbf{skip}; \\
&\quad (2) \quad \mathbf{await}\ c.\mathbf{sig} = c.\mathbf{ack}; \\
&\quad \quad c.\mathbf{data} := i; \\
&\quad (3) \quad c.\mathbf{sig} := \neg c.\mathbf{sig} \\
\mathbf{P}_2 &::= \mathbf{while\ true\ do} \\
&\quad (0) \quad \mathbf{await}\ c.\mathbf{sig} \neq c.\mathbf{ack}; \\
&\quad \quad o := c.\mathbf{data}; \\
&\quad (1) \quad c.\mathbf{ack} := \neg c.\mathbf{sig} \\
&\quad (2) \quad l_2 : \mathbf{skip} \\
\mathbf{Env} &::= \square (i'' = i' \wedge o'' = o' \wedge c'' = c' \wedge l_1'' = l_1' \wedge l_2'' = l_2')
\end{aligned}$$

Two processes \mathbf{P}_1 and \mathbf{P}_2 communicate with simple handshaking. The property is to verify that always, if data has been produced, it is eventually sent and received. The proof obligation differs from Chapter 1 in that the procedures `send` and `receive` have been expanded. Furthermore, an environment assumption \mathbf{Env} has been added to ensure that the system variables are not modified by further processes; in other words, the system with the sending and the receiving process is closed.

Markers $(0), (1), \dots$ are used to (informally) refer to a system configuration. The initial system configuration consists of three digits 000, where the first digit refers to the configuration of the first process, the second to the configuration of the second process, and the third to the configuration of the property to verify.

7.3.1 Proof without abstraction

Verifying the example without further abstraction results in the proof graph of Fig. 7.1. (The graph only illustrates steps, sequencing, and generalization. Details for the single steps as well as (simultaneous) induction rules have been omitted as they have already been discussed in previous Chapters 4 and 6.) The proof roughly falls into two parts: the lower part is concerned with the verification of the original property

$$(0) \square (l_1' \neq l_1 \wedge D = i \rightarrow (1) \diamond (l_2' \neq l_2 \wedge o = D))$$

while the part on top considers sub-formula

$$(1) \diamond (l_2' \neq l_2 \wedge o = D) .$$

Starting at configuration 000, the first step either executes the first or the second process and arrives at labels 100 and 010. Execution continues until the system loops and induction can be applied.

The invariants for the different states where (simultaneous) induction is applied mostly equal the PL formulas describing the current state. Only once, the invariant is more general. The

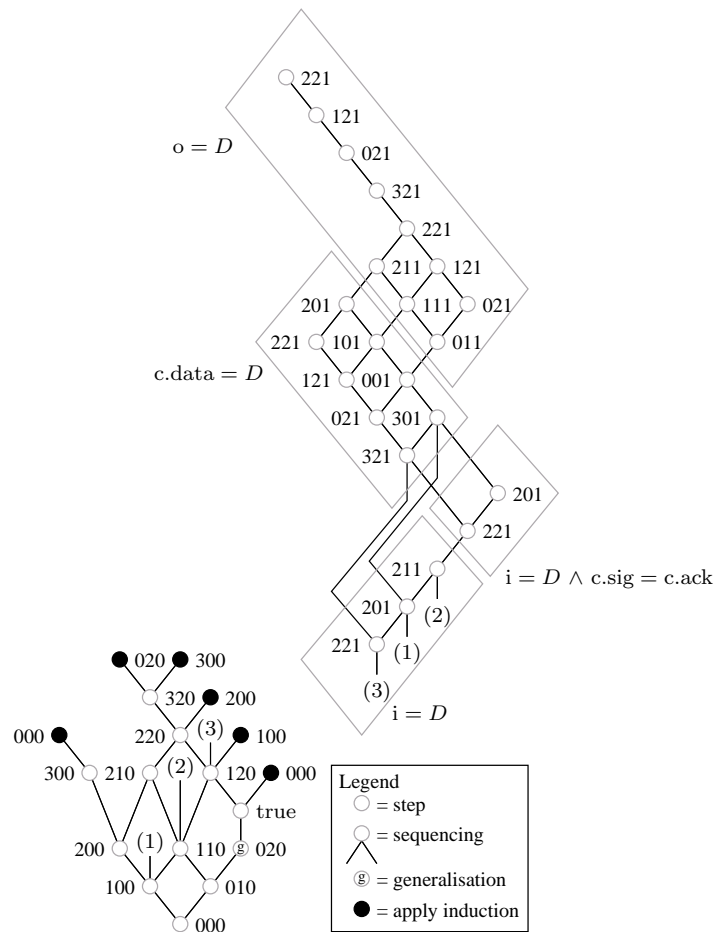


Figure 7.1: Proof graph for handshake example without abstraction

state is explicitly shown in the proof graph as a generalization step. In step 020, the second process has received data from the channel and the PL formulas are

$$o = c.data \wedge c.sig \neq c.ack$$

i.e., currently, o equals the data on the channel and the channel is unoccupied. It is now possible that the first process loops and writes different data to the channel. Thus, after returning to configuration 020, the channel is now occupied and the data is different from o . Therefore, we generalize with $Inv \equiv true$.

If the first process progresses from 1 to 2 (e.g. in configuration 100), the label l_1 toggles and the precondition of the leads-to property is triggered. Thus, we receive two cases: in one branch the always operator is further considered (configuration 200) while in the other branch the eventually property must be established (201). In total, there are three triggers of the leads-to property (1), (2), and (3).

The eventually property $\diamond l_2 \neq l_2 \wedge o = D$ is considered in the upper part of the proof. The strategy is to simply execute the system until a state is reached where the property is satisfied. The proof falls into four parts, the first part where D has not yet been sent ($i = D$), the second part where D has not yet been sent and the channel is unoccupied ($c.sig = c.ack$), the third part where D has been written to the channel ($c.data = D$), and the fourth part, where the data has been received ($o = D$) and it remains to trigger label l_2 .

Overall, the proof is rather straightforward. Symbolic execution, sequencing, and induction can be automated, and only a single invariant in configuration 020 must be manually provided. However, the proof is rather large visiting 11 different configurations in the lower part, and passing through 23 states in the upper part of the proof. We will now try to reduce the number of intermediate states with abstraction.

7.3.2 Abstraction

In order to shorten the proof, we need to ensure that execution of the two processes loops after less steps. One approach is to define for the two processes a static transition system by introducing an additional variable which serves as a program counter, and to generalize the value of this program counter within the proof.

Property *sts-1* of Fig. 7.2 relates the first process P_1 on the left hand side of the implication to a static transition system STS_1 on the right hand side. A local variable p serves as a program counter. The commands of the first process are turned into transitions. Every transition consists of an activation condition and an action. The activation condition refers to the program counter and the action updates the program counter accordingly. For example, command $i := ?$ corresponds to the transition

$$\mathbf{await} \ p = 0; \{i := ?; p := 1\}$$

The transition is activated, if and only if $p = 0$. It assigns a random value to i and updates the program counter to 1. The assignments are combined in an atomic block to ensure that they

where

$$sts-1: \mathbf{P}_1 \rightarrow \mathbf{var} \ p = 0 \ \mathbf{in} \ \mathbf{STS}_1$$

$$\begin{aligned} \mathbf{STS}_1 &::= \mathbf{while} \ \mathbf{true} \ \mathbf{do} \\ &\quad \mathbf{await} \ p = 0; \\ &\quad \{i := ?; p := 1\} \\ &\quad \parallel \mathbf{await} \ p = 1; \\ &\quad \quad l_1 : p := 2 \\ &\quad \parallel \mathbf{await} \ p = 2 \wedge c.\mathbf{sig} = c.\mathbf{ack}; \\ &\quad \quad \{c.\mathbf{data} := i; p := 3\} \\ &\quad \parallel \mathbf{await} \ p = 3; \\ &\quad \quad \{c.\mathbf{sig} := \neg c.\mathbf{ack}; p := 0\} \end{aligned}$$

Figure 7.2: Static transition system for first process in handshake example

where

$$sts-2: \mathbf{P}_2 \rightarrow \mathbf{var} \ q = 0 \ \mathbf{in} \ \mathbf{STS}_2$$

$$\begin{aligned} \mathbf{STS}_2 &::= \mathbf{while} \ \mathbf{true} \ \mathbf{do} \\ &\quad \mathbf{await} \ q = 0 \wedge c.\mathbf{sig} \neq c.\mathbf{ack}; \\ &\quad \{o := c.\mathbf{data}; q := 1\} \\ &\quad \parallel \mathbf{await} \ q = 1; \\ &\quad \quad \{c.\mathbf{ack} := \neg c.\mathbf{sig}; q := 2\} \\ &\quad \parallel \mathbf{await} \ q = 2; \\ &\quad \quad l_2 : q := 0 \end{aligned}$$

Figure 7.3: Static transition system for second process in handshake example

are executed in a single step. The choice operator, which combines the different transitions, ensures that a transition for which the await condition is satisfied is executed. Execution of transitions loops infinitely often.

The second process can also be turned into a static transition system \mathbf{STS}_2 with a program counter q (see property *sts-2* of Fig. 7.3). Congruence rules for interleaving can now be applied to replace the processes with the static transition system in the original proof obligation.

$$\begin{array}{c} \frac{\begin{array}{l} \vdash \ \mathbf{var} \ p = 0 \ \mathbf{in} \ \mathbf{STS}_1 \ \parallel \ \mathbf{var} \ q = 0 \ \mathbf{in} \ \mathbf{STS}_2 \ \wedge \ \mathbf{Env} \\ \rightarrow \square \ (l'_1 \neq l_1 \wedge D = i \rightarrow (\diamond l'_2 \neq l_2 \wedge o = D)) \end{array}}{\vdash \ \mathbf{var} \ p = 0 \ \mathbf{in} \ \mathbf{STS}_1 \ \parallel \ \mathbf{P}_2 \ \wedge \ \mathbf{Env}} \quad sts-2 \\ \frac{\vdash \ \mathbf{var} \ p = 0 \ \mathbf{in} \ \mathbf{STS}_1 \ \parallel \ \mathbf{P}_2 \ \wedge \ \mathbf{Env}}{\rightarrow \square \ (l'_1 \neq l_1 \wedge D = i \rightarrow (\diamond l'_2 \neq l_2 \wedge o = D))} \quad sts-1 \\ \vdash \ \mathbf{P}_1 \ \parallel \ \mathbf{P}_2 \ \wedge \ \mathbf{Env} \\ \rightarrow \square \ (l'_1 \neq l_1 \wedge D = i \rightarrow (\diamond l'_2 \neq l_2 \wedge o = D)) \end{array}$$

In order to generalize the local values of p and q , we introduce additional dynamic variables p and q with rule *var xtrct* (see Appendix B.8.4).

$$\begin{array}{c}
\vdash \quad p = 0 \wedge q = 0 \wedge \mathbf{STS}'_1 \parallel \mathbf{STS}'_2 \wedge \mathbf{Env} \\
\rightarrow \square (l'_1 \neq l_1 \wedge D = i \rightarrow \diamond (l'_2 \neq l_2 \wedge o = D)) \\
\hline
\vdash \quad (\exists p. p = 0 \wedge \mathbf{var} \ p = p \ \mathbf{in} \ (\square p' = p \wedge \mathbf{STS}_1)) \\
\parallel (\exists q. q = 0 \wedge \mathbf{var} \ q = q \ \mathbf{in} \ (\square q' = q \wedge \mathbf{STS}_2)) \\
\wedge \mathbf{Env} \\
\rightarrow \square (l'_1 \neq l_1 \wedge D = i \rightarrow \diamond (l'_2 \neq l_2 \wedge o = D)) \\
\hline
\vdash \quad (\exists p. p = 0 \wedge \mathbf{var} \ p = p \ \mathbf{in} \ (\square p' = p \wedge \mathbf{STS}_1)) \\
\parallel \mathbf{var} \ q = 0 \ \mathbf{in} \ \mathbf{STS}_2 \\
\wedge \mathbf{Env} \\
\rightarrow \square (l'_1 \neq l_1 \wedge D = i \rightarrow \diamond (l'_2 \neq l_2 \wedge o = D)) \\
\hline
\vdash \quad \mathbf{var} \ p = 0 \ \mathbf{in} \ \mathbf{STS}_1 \parallel \mathbf{var} \ q = 0 \ \mathbf{in} \ \mathbf{STS}_2 \wedge \mathbf{Env} \\
\rightarrow \square (l'_1 \neq l_1 \wedge D = i \rightarrow (\diamond l'_2 \neq l_2 \wedge o = D))
\end{array}
\begin{array}{l}
\text{simp} \\
\text{var xtrct} \\
\text{var xtrct}
\end{array}$$

where

$$\begin{aligned}
\mathbf{STS}'_1 &::= \mathbf{var} \ p = p \ \mathbf{in} \ (\square p' = p' \wedge \mathbf{STS}_1) \\
\mathbf{STS}'_2 &::= \mathbf{var} \ q = q \ \mathbf{in} \ (\square q' = q' \wedge \mathbf{STS}_2) .
\end{aligned}$$

This sequent is now subject to generalization: instead of starting the proof in system configuration 000, we generalize the configuration to a more general state where the configuration of the first and the second process is arbitrary. We only require that the two processes do not both reside in their critical section. This is expressed in the following invariant:

$$(p = 3 \rightarrow c.\text{sig} = c.\text{ack}) \wedge (q = 1 \rightarrow c.\text{sig} \neq c.\text{ack})$$

The first process resides in its critical section ($p = 3$) only, if the channel is currently unoccupied ($c.\text{sig} = c.\text{ack}$), and similarly for the second process.

If the more general system is executed, induction can be applied after a single step. This is depicted in the proof graph of Figure 7.4. The system in configuration $pq0$ immediately loops. Thus, the 11 different configurations of the lower part of the proof of Figure 7.1 collapse to a single configuration, and it only remains to establish the eventually condition $\diamond l'_2 \neq l_2 \wedge o = D$ in premise $2q1$.

The sub-proof for establishing the eventually condition again falls into four parts.

In the beginning, $i = D$ and the first process is trying to write D into the channel. Either the channel is immediately unoccupied ($c.\text{sig} = c.\text{ack}$) or not ($c.\text{sig} \neq c.\text{ack}$). In the latter case, an additional liveness property is required for the second process

$$\text{sts-2-progress-1: } \mathbf{STS}'_2 \rightarrow \diamond c'.\text{sig} = c'.\text{ack}$$

to ensure that the second process always eventually releases the channel. This property holds

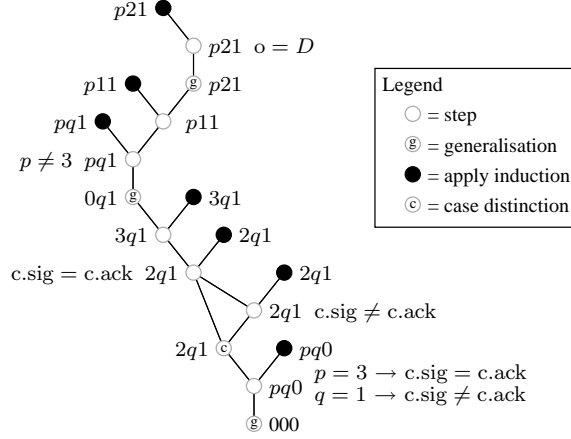


Figure 7.4: Abstract proof graph for handshake example

independent from the environment relation! The liveness property is introduced as follows:

$$\begin{array}{c}
 \vdash \quad p = 2 \wedge c.\text{sig} \neq c.\text{ack} \\
 \wedge \mathbf{STS}'_1 \parallel (\mathbf{STS}'_2 \wedge \diamond c'.\text{sig} = c'.\text{ack}) \wedge \mathbf{Env} \\
 \rightarrow \diamond (l'_2 \neq l_2 \wedge o = D) \\
 \hline
 \vdash \quad p = 2 \wedge c.\text{sig} \neq c.\text{ack} \\
 \wedge \mathbf{STS}'_1 \parallel (\mathbf{STS}'_2 \wedge \mathbf{STS}'_2) \wedge \mathbf{Env} \\
 \rightarrow \diamond (l'_2 \neq l_2 \wedge o = D) \\
 \hline
 \vdash \quad p = 2 \wedge c.\text{sig} \neq c.\text{ack} \\
 \wedge \mathbf{STS}'_1 \parallel \mathbf{STS}'_2 \wedge \mathbf{Env} \\
 \rightarrow \diamond (l'_2 \neq l_2 \wedge o = D)
 \end{array}
 \quad \begin{array}{l}
 \text{sts-2-progress-1} \\
 \\
 \text{con dpl}
 \end{array}$$

With rules *ev live* and *ilv live 2*, the eventually property can be lifted to top level to receive

$$\begin{array}{c}
 \vdash \quad p = 2 \wedge c.\text{sig} \neq c.\text{ack} \\
 \wedge \diamond l \wedge \mathbf{STS}'_1 \parallel l :: (\mathbf{STS}'_2 \wedge \neg l \text{ until } c'.\text{sig} = c'.\text{ack}) \wedge \mathbf{Env} \\
 \rightarrow \diamond (l'_2 \neq l_2 \wedge o = D)
 \end{array}$$

The second process ensures that l is false until $c'.\text{sig} = c'.\text{ack}$. Furthermore, l is false, if transitions of the first process are executed. Thus, inducing over the number of steps it takes to satisfy l is to induce over the number of steps it takes for the second process to release the channel. After executing the next step, the channel has either been released or induction can be applied.

For the second part of the proof $i = D$ and $c.\text{sig} = c.\text{ack}$, and the next transition of the first process will write D to the channel. However, an arbitrary number of transitions of the second process can be executed first. Because interleaving is fair, we can induce over the number of

transitions preceding a transition of the first process.

$$\begin{array}{c}
\vdash \quad p = 2 \wedge \text{c.sig} = \text{c.ack} \wedge \diamond l \wedge l :: \mathbf{STS}'_1 \parallel \mathbf{STS}'_2 \wedge \mathbf{Env} \\
\rightarrow \diamond (l'_2 \neq l_2 \wedge o = D) \\
\hline
\vdash \quad p = 2 \wedge \text{c.sig} = \text{c.ack} \wedge (\exists l. \diamond l \wedge l :: \mathbf{STS}'_1 \parallel \mathbf{STS}'_2) \wedge \mathbf{Env} \\
\rightarrow \diamond (l'_2 \neq l_2 \wedge o = D) \\
\hline
\vdash \quad p = 2 \wedge \text{c.sig} = \text{c.ack} \wedge \mathbf{STS}'_1 \parallel \mathbf{STS}'_2 \wedge \mathbf{Env} \\
\rightarrow \diamond (l'_2 \neq l_2 \wedge o = D)
\end{array}
\begin{array}{l}
\text{simp} \\
\text{ilv fair 1}
\end{array}$$

After the next step, we either arrive at configuration $3q1$ and $\text{c.data} = D$ or the second process has been executed and induction can be applied. In the same way, an additional step of the first process can be executed to invert the signal c.sig and to arrive with configuration $0q1$. The first process has successfully written D to the channel.

For the third part of the proof, the second process must now be considered, and the configuration of the first process can be abstracted. It is only important that the latter does not reside in its critical section ($p \neq 3$). Property

$$\text{sts-2-progress-2: } \mathbf{STS}'_2 \quad \rightarrow \quad \diamond q = 0$$

guarantees that the second process eventually receives data from the channel. The eventually property can be used for induction as shown above. Thus, we either apply induction or arrive with configuration $p11$ and $o = D$. Another transition of process two releases the channel. Thus, in configuration $p21$, $\text{c.sig} = \text{c.ack}$.

In the fourth part of the proof it remains to trigger label l_2 . As process one is allowed to arbitrarily manipulate the channel, we generalize the state such that only $o = D$ remains. The next transition of the second process toggles l_2 . If the label toggles, the proof is finished. Otherwise, a transition of process one has been executed. In the latter case, because interleaving is fair, induction can be applied.

The final proof for establishing the eventually property $\diamond (l'_2 \neq l_2 \wedge o = D)$ considers five different configurations instead of passing through 23 configurations in Figure 7.1. However, six different inductions are necessary to ensure progress. It remains to prove properties *sts-1*, *sts-2*, *sts-2-progress-1*, and *sts-2-progress-2*. Proofs for these properties are straightforward as the single processes are sequential programs without further interleaving.

7.4 Verification diagrams

The approach in the example is very similar to the use of verification diagrams in STeP [21]. “Verification diagrams are a visual representation of a proof; a well-formed verification diagram represents a set of verification conditions which are sufficient to establish a given formula.” A verification diagram basically is a state transition system with first order formulas labeling the different states. A state in a verification diagram represents the

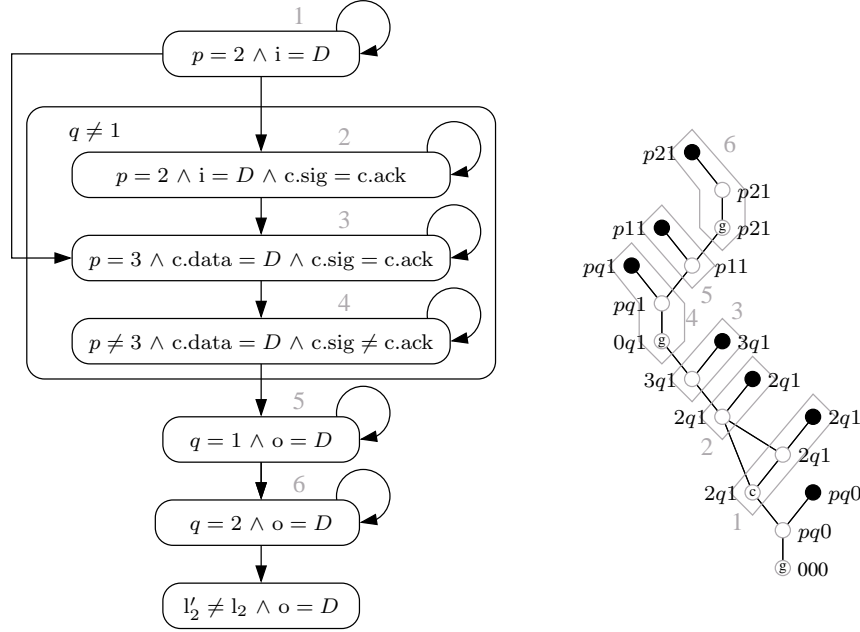


Figure 7.5: Verification diagram for handshake

set of system states which satisfy the associated formula. Thus, the verification diagram is an abstract system description.

Figure 7.5 contains a verification diagram on the left for the second part of the example proof, i.e., the verification of the eventually property $\diamond (l'_2 \neq l_2 \wedge o = D)$. The diagram closely corresponds to the proof of Fig. 7.4 starting with configuration $2q1$. The correspondence of states to proof steps is illustrated on the right of Figure 7.5. The first state represents the three proof steps with configuration $2q1$ where the channel is not yet guaranteed to be unoccupied; the second state represents the two proof steps with configuration $2q1$ where in addition $c.sig = c.ack$; in the same manner, the third to sixth state correspond to a group of steps. Thus, the verification diagram visualizes our example proof in an intuitive style.

The verification diagram of Fig. 7.5 only considers the second part of the leads-to property. To verify the complete property, additional verification conditions must be solved in STeP. Different verification diagrams are tailored to the verification of special classes of properties: invariants $\square \varphi_{\mathbf{PL}}$, liveness properties $\diamond \varphi_{\mathbf{PL}}$, leads-to properties $\square \varphi_{\mathbf{PL}} \rightarrow \diamond \psi_{\mathbf{PL}}$, and wait-for properties $\varphi_{\mathbf{PL}}^1 \text{ unless } \dots \text{ unless } \varphi_{\mathbf{PL}}^n$. In theory, so called generalized verification diagrams [30] can be applied to arbitrary temporal formulas; however, a generalized diagram is difficult to apply in practice.

Our strategy of proof, on the other hand, is to symbolically execute the system inde-

pendent from the property to verify, but proofs are the larger the more complicated the property. Therefore, abstraction is essential also for symbolic execution. It would be promising to investigate a combination of symbolic execution and verification diagrams. Symbolic execution is intuitive and can be applied to arbitrary properties while verification diagrams are well suited to visualize abstractions. A more detailed comparison of symbolic execution and verification diagrams can be found in [29].

7.5 Conclusion

The use of double primed variables to model environment transitions is the key to compositionality in our logic. On every trace, system and environment transitions alternate. To verify $\vdash \varphi_1 \rightarrow \varphi_2$ is to verify the property for every possible environment. Because our semantics of interleaving is compositional, an interleaved process φ_1 can be safely replaced by a more abstract property φ_2 .

The strategy of abstraction in our example is closely related to verification diagrams in STeP. As has been discussed in Section 7.4 an integration of symbolic execution and verification diagrams should be further examined.

In practice, more complex properties of single components only hold under additional assumptions. In TLA, so called assumption-commitment specifications are used to formalize properties which rely on a restricted behavior of the environment in a certain style [1]. In our logic, it is possible to express environment assumptions as relation between primed and double primed variables. It should be possible to carry over results concerning assumption-commitment (or rely-guarantee) specifications to our approach.

Chapter 8

Concluding Remarks

We have defined a temporal logic where operators for parallel programs and temporal logic can be arbitrarily nested. We support interleaving with explicit blocking, non-deterministic choice, and others. Most important, the semantics of all of the operators are compositional. Thus, systems can be abstracted and proofs can be decomposed. This ensures that our strategy of proof can be applied to the verification of large, concurrent systems.

Symbolic execution of parallel programs is indeed an intuitive strategy to deduce temporal properties. Our induction rules are general enough to verify all kinds of properties. For liveness and safety properties, the strategy is the same: execute the system until it either terminates or loops. Use induction for the latter case. In order to avoid exponential growth of proof size, we have introduced the strategy of sequencing, which is again applicable for arbitrary temporal properties.

The strategy has been implemented in KIV, and the implementation has shown that proofs can indeed be automated to a large extent. Figure 8.1 gives an example proof where one-bounded overtaking for the bakery algorithm to implement mutual exclusion has been verified. Purple nodes represent step execution, red nodes with blue arrows visualize induction, and light green arrows connect premises which have been sequentialized. Black nodes simplify a given premise. All of the proof has been automatically constructed, except for two proof steps where the PL formulas for the current state had to be generalized with an invariant.

We have applied our method of proof to several examples. As has already been mentioned, we have verified the bakery algorithm for mutual exclusion. A simple handshaking protocol has been examined. Furthermore, a parallel program for the calculation of binomial coefficients has been verified [9]. This case study involved complex PL reasoning in the underlying domain. The largest case study has been the application of our methods to the verification of medical guidelines [15].

Our approach has been mainly inspired by symbolic execution of sequential programs

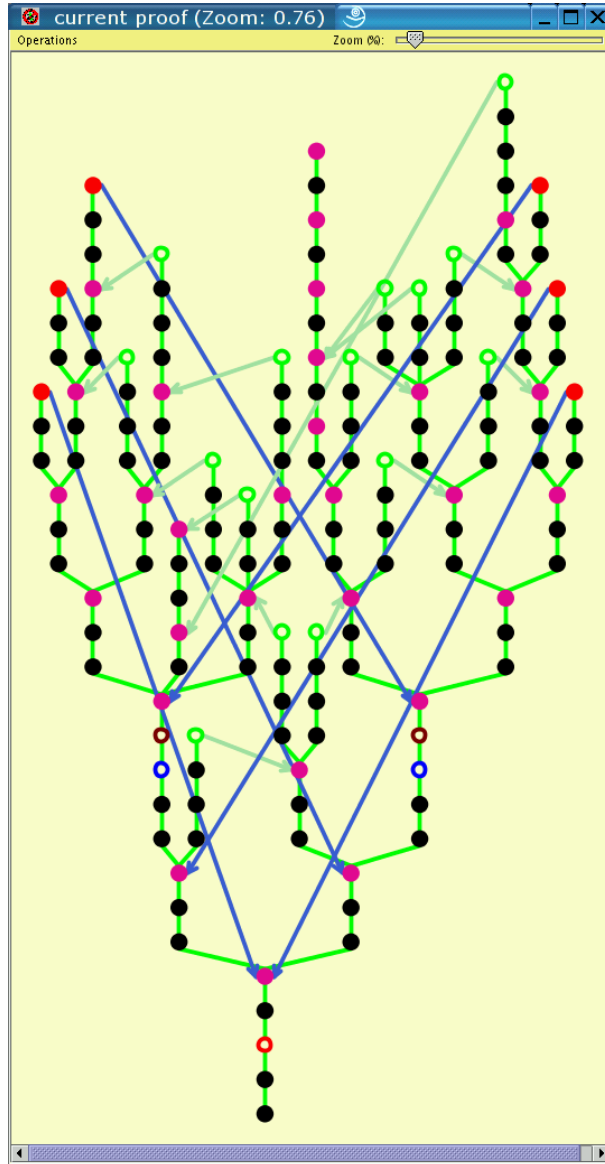


Figure 8.1: Example proof in KIV

in Dynamic Logic (DL), the composition of concurrent systems in the Temporal Logic of Actions (TLA), and the verification of parallel programs with the Stanford Temporal prover (STeP). Section 8.1 compares our logic and proof method to these approaches. Finally, Section 8.2 gives an outlook into the future.

8.1 Comparison

8.1.1 Dynamic Logic

In Dynamic Logic, sequential programs are symbolically executed to establish a property for the final state of execution. We have shown how to apply the strategy of symbolic execution also to concurrent systems. Furthermore, we show temporal properties for the whole trace of execution.

The rules of Fig. 3.1 in Sect. 3.1 are based on a sequent calculus. In a sequent calculus, rules are only applied to the top level operators. In our approach, we rewrite sub-formulas to execute temporal formulas. The example of Section 4.5.4 has shown that it is rather complicated to rewrite sub-formulas first and afterwards to rewrite the surrounding operators. In Dynamic Logic, the example

$$\vdash [\mathbf{begin} \ n := 0; m := 1 \ \mathbf{end}; n := 1]_{DL} \varphi$$

would be executed as follows:

$$\frac{\frac{\frac{n = 0 \vdash [m := 1]_{DL}([n := 1]_{DL} \varphi)}{\vdash [n := 0]_{DL}([m := 1]_{DL}([n := 1]_{DL} \varphi))} \text{asg_r}}{\vdash [n := 0; m := 1]_{DL}([n := 1]_{DL} \varphi)} \text{norm}}{\vdash [\mathbf{begin} \ n := 0; m := 1 \ \mathbf{end}; n := 1]_{DL} \varphi} \text{norm}$$

In a number of normalizations steps, compositions are eliminated and sequences of box operators are introduced. Finally the assignment is the top level construct of the leading box operator and rule *asg_r* is applied. This approach is possible only because sequences of box operators and program composition are very similar. The semantics of

$$[\alpha]_{DL}[\beta]_{DL} \varphi$$

can be read as follows: "if α terminates, then β is executed. If β finally terminates then φ holds in the final state". For parallel programs the property of interest has to hold not only in the final state but for the trace in total. This prevents reducing composition to sequences of box operators and thus lifting the first program to top level. Also, there are other operators which cannot directly be executed, e.g., interleaving. Interleaving and composition can be nested, for example

$$(n := 0; m := 1) \parallel (n := 1 \parallel n := 2).$$

Before the interleaving operator can be rewritten, the sub-formulas need to be rewritten first. This is why we have followed a more generic approach which allows for application of rules to sub-formulas. Nevertheless, symbolic execution can be automated to a large extent and the advantage of Dynamic Logic being an intuitive and highly automatic strategy of proof carries over to the execution of interleaved parallel programs.

Executing interleaved parallel programs regularly leads to a large number of possible runs, which is not the case for the execution of sequential programs with a single thread of execution. As a remedy to this, we have introduced the strategy of sequencing in Section 5. If restricted to the verification of post conditions for sequential programs, our approach gives similar proofs, if compared to Dynamic Logic. With the system operators of Sect. 2.2.10 it is possible to fully embed the box and diamond operator of Dynamic Logic in our logic: for a sequential program α and a post condition φ_{PL} , we receive

$$\begin{aligned} \vdash_{\text{DL}} [\alpha]_{\text{DL}} \varphi_{\text{PL}} & \text{ iff } \vdash [\alpha] \square (\mathbf{last} \rightarrow \varphi_{\text{PL}}) \\ \vdash_{\text{DL}} \langle \alpha \rangle_{\text{DL}} \varphi_{\text{PL}} & \text{ iff } \vdash \langle \alpha \rangle \diamond (\mathbf{last} \wedge \varphi_{\text{PL}}) \end{aligned}$$

The embedding of Dynamic Logic has already been investigated in [31].

8.1.2 Temporal Logic of Actions

In Temporal Logic of Actions, the environment is considered in stuttering steps as has been explained in Section 1.8.3. The environment is part of every system model which is formalized as a temporal formula in normal form; environment transitions are considered on object level. We have integrated environment transitions as relation between primed and double primed variables on a semantics level. As a consequence, formulas need not adhere to a fixed normal form, and we are able to define a rich language with complex operators for parallel programs such as interleaving and nondeterministic choice and still preserve compositionality.

In TLA, modules \mathcal{M}_1 and \mathcal{M}_2 are composed with simple conjunction $\mathcal{M}_1 \wedge \mathcal{M}_2$. Note that in a conjunction of modules, the single module can be trivially abstracted. For example, instead of verifying

$$\mathcal{M}_1 \wedge \mathcal{M}_2 \rightarrow \varphi$$

i.e., the composition of two components satisfies φ , the component \mathcal{M}_1 can be replaced by an abstract component \mathcal{M}'_1 , and the proof can be decomposed into two properties

1. $\mathcal{M}_1 \rightarrow \mathcal{M}'_1$, and
2. $\mathcal{M}'_1 \wedge \mathcal{M}_2 \rightarrow \varphi$.

The soundness of decomposition directly follows from propositional logic. In our logic, the proof obligation

$$\varphi_1 \parallel \psi \rightarrow \chi$$

can also be decomposed into the same properties

1. $\varphi_1 \rightarrow \varphi_2$, and
2. $\varphi_2 \parallel \psi \rightarrow \chi$.

The decomposition is sound, because – thanks to double primed variables – every possible environment is accounted for in the verification of $\varphi_1 \rightarrow \varphi_2$. Therefore, it should be possible to carry over results of conjoining specifications in TLA to our approach.

There is, however, a subtle difference in verifying properties in TLA and in our approach. If $\mathcal{M}_1 \rightarrow \Box \varphi$ holds, then both system and every environment transitions satisfy φ ; the stuttering steps of \mathcal{M}_1 are additional transitions of the overall system. In conjunction with a second module $\mathcal{M}_1 \wedge \mathcal{M}_2$, the property still holds: $\mathcal{M}_1 \wedge \mathcal{M}_2 \rightarrow \Box \varphi$. In our approach, if $\varphi_1 \rightarrow \Box \varphi_2$ holds, then φ_2 is satisfied by every system transition for every possible environment; however, property φ_2 does not necessarily hold for an interleaved system

$$\varphi_1 \parallel \psi \stackrel{?}{\rightarrow} \Box \varphi_2 .$$

The first process can only be replaced by the property: $(\Box \varphi_2) \parallel \psi$ and it remains to verify

$$(\Box \varphi_2) \parallel \psi \rightarrow \Box \varphi_2 .$$

This difference must be further explored.

8.1.3 The STeP approach

Our language to define parallel programs is closely related to the specification language of reactive systems in STeP. Our semantics for interleaving and nondeterministic choice has been inspired by [19], which is also fundamental to STeP. In STeP, system descriptions are compiled into a fair transition system in advance to verification, whereas we maintain the original description in our logic; the system is compiled to a transition system “on the fly” while executing transitions. Maintaining the original description is in our opinion more intuitive than reasoning in a different formalism.

The verification rules of STeP focus on single transitions. For example, rule INV establishes a safety property $\Box \varphi$ for a given transition system M as follows:

$$\frac{\begin{array}{l} \text{I1. } \varphi \rightarrow \psi \\ \text{I2. } \Theta \rightarrow \psi \\ \text{I3. } \{\psi\}\tau\{\psi\} \text{ for each } \tau \in \mathcal{T} \end{array}}{M \models \Box \varphi} \text{ INV}$$

where Θ is the initial condition of M , and \mathcal{T} is the set of transitions. Formula ψ is an invariant which is possibly more general than the safety condition φ ; the invariant must be general enough to be maintained by all of the transitions τ . In our approach, we use induction to establish $\Box \varphi$ as has been described in Chapter 6: the system is executed until a system configuration is reached which has already been encountered. In this case,

induction is applied. We also need to provide an invariant; however, the invariant is only required to hold before and after the execution of a complete loop, and the loop may consist of several transitions. As a consequence, our invariants are often smaller than invariants in STeP. This has been investigated in [9].

STeP offers an interesting concept to visualize proofs: verification diagrams. A detailed comparison of symbolic execution with abstraction and verification diagrams has already been given in Section 7.4. Summarizing, it would be promising to investigate a combination of symbolic execution and verification diagrams: symbolic execution, on the one hand, is intuitive and can be applied to arbitrary properties while verification diagrams, on the other hand, are well suited to visualize abstractions.

STeP also integrates model checking, various decision procedures, and offers algorithms to automatically generate invariants. Furthermore, special support for the verification of real time systems is given. An integration with our strategy of symbolic execution has not yet been considered. A first example to verify real time systems with symbolic execution is given in [29].

8.2 Outlook

The strategy of symbolic execution can be extended to include other formalisms besides interleaved parallel programs. [32] shows how to integrate Statemate state-charts into our framework. In [5], rules to execute UML state-charts were successfully defined. Further operators have been examined, including strong fair interleaving (see [29]), synchronous execution and interrupts. As part of the European project Protocure, we have applied the strategy to the execution of Asbru, an planning language for modeling medical guidelines (see [15]). For all of these integrations, a new type of formula had to be defined, and a set of rules to rewrite the formalisms to normal form, i.e., to execute the first transitions, had to be implemented. Our induction strategy, sequencing, and abstraction remained unchanged.

We have only partially considered system operators $[\varphi] \psi$, and $\langle \varphi \rangle \psi$. A thorough integration of these operators remains open. Especially for the diamond operator, rules for induction and sequencing have to be defined. With the diamond operator, it would then be possible to verify the existence of paths satisfying a given property.

With the system operators, we have partially carried over our strategy of proof to branching time logics. It remains an open issue whether symbolic execution can be applied to CTL* in general. Furthermore, support for past tense operators and real-time applications would be interesting to construct.

We have only sketched an argument for our proof method to be (partially) complete. A formal proof still needs to be constructed. Furthermore, we did not consider all of our operators while proving the coincidence lemma and other properties of our logic: proofs for nondeterministic choice remain open. Instead, we have focused on the suitability of the set of rules in practice: most of the rules can be automatically applied, and all of the

rules ensure that the premises remain intuitive. Nevertheless, the set of rules should be refined with further applications.

The implementation in KIV must be considered a prototype: step execution is slow, because it is implemented as a set of basic rewrite rules. A faster algorithm would be obtained, if an algorithm for rewriting a formula to normal form in a single step would be constructed. Interactive verification would be further improved, if the current system configuration would be represented in a more intuitive style. For the execution of state-charts, a graphical representation of the current state would be possible. For parallel programs, the visualization of source code with arrows as program counters would be more intuitive than consuming the program during execution. In the long run, symbolic execution could be as comfortable as debugging programs in a state-of-the-art integrated development environment.

Appendix A

Rewriting

A.1 Basic rules

A.1.1 Top level rules

Rules to apply to top level formula. Rule *true* closes a premise, rule *ax* exploits a precondition, axiom or lemma to close a premise, rule *con* generates two premises for a conjunction, and rule *all* eliminates universal quantification.

$$\frac{}{\Gamma \vdash \text{true}} \text{true} \quad \frac{}{\Gamma_1, \varphi, \Gamma_2 \vdash \varphi} \text{ax}$$
$$\frac{\Gamma \vdash \varphi_1 \quad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2} \text{con} \quad \frac{\Gamma \vdash \varphi[v_0/v]}{\Gamma \vdash \forall v. \varphi} \text{all}$$

where $v_0 \in \mathbf{Y}$ is fresh with respect to $(\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(\Gamma)$.

A.1.2 Basic rewrite rules

Rule *rewrite* initiates rewriting. Rules *close** finish rewriting of a sub expression and propagate the result.

$$\frac{\Gamma \vdash_{\mathbf{c}} \varphi \supset \psi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi} \text{rewrite}$$
$$\frac{}{\Gamma \vdash_{\mathbf{c}} \varphi \supset \varphi} \text{close}^{\supset} \quad \frac{}{\Gamma \vdash_{\mathbf{c}} \varphi \subset \varphi} \text{close}^{\subset} \quad \frac{}{\Gamma \vdash_{\mathbf{c}} e = e} \text{close}^{\equiv}$$

A.1.3 Derived axiom rules

Rules to exploit preconditions from the context to prove or to contradict the current formula.

$$\frac{\Gamma_1, \varphi, \Gamma_2 \vdash_{\mathbf{c}} \text{true} \supset \chi}{\Gamma_1, \varphi, \Gamma_2 \vdash_{\mathbf{c}} \varphi \supset \chi} \text{ ax true}^{\supset} \quad \frac{\Gamma_1, \varphi, \Gamma_2 \vdash_{\mathbf{c}} \text{true} \subset \chi}{\Gamma_1, \varphi, \Gamma_2 \vdash_{\mathbf{c}} \varphi \subset \chi} \text{ ax true}^{\subset}$$

$$\frac{\Gamma_1, \varphi, \Gamma_2 \vdash_{\mathbf{c}} \text{true} = \chi}{\Gamma_1, \varphi, \Gamma_2 \vdash_{\mathbf{c}} \varphi = \chi} \text{ ax true}^{\text{=}}$$

$$\frac{\Gamma_1, \neg \varphi, \Gamma_2 \vdash_{\mathbf{c}} \text{false} \supset \chi}{\Gamma_1, \neg \varphi, \Gamma_2 \vdash_{\mathbf{c}} \varphi \supset \chi} \text{ ax false}^{\supset} \quad \frac{\Gamma_1, \neg \varphi, \Gamma_2 \vdash_{\mathbf{c}} \text{false} \subset \chi}{\Gamma_1, \neg \varphi, \Gamma_2 \vdash_{\mathbf{c}} \varphi \subset \chi} \text{ ax false}^{\subset}$$

$$\frac{\Gamma_1, \neg \varphi, \Gamma_2 \vdash_{\mathbf{c}} \text{false} = \chi}{\Gamma_1, \neg \varphi, \Gamma_2 \vdash_{\mathbf{c}} \varphi = \chi} \text{ ax false}^{\text{=}}$$

A.2 Rewrite rules

Rules to rewrite top level expression.

A.2.1 Simple rewrite rules

Let

$$\begin{aligned} \text{lem}_1: & \quad \varphi_1 \quad \leftrightarrow \quad \varphi_2 \\ \text{lem}_2: & \quad e_1 \quad = \quad e_2 \end{aligned}$$

then

$$\frac{\Gamma \vdash \text{lem}_1 \quad \Gamma \vdash_{\mathbf{c}} \varphi_2 \supset \psi}{\Gamma \vdash_{\mathbf{c}} \varphi_1 \supset \psi} \text{ rw}^{\supset} \quad \frac{\Gamma \vdash \text{lem}_1 \quad \Gamma \vdash_{\mathbf{c}} \varphi_2 \subset \psi}{\Gamma \vdash_{\mathbf{c}} \varphi_1 \subset \psi} \text{ rw}^{\subset}$$

$$\frac{\Gamma \vdash \text{lem}_2 \quad \Gamma \vdash_{\mathbf{c}} e_2 = e}{\Gamma \vdash_{\mathbf{c}} e_1 = e} \text{ rw}^{\text{=}}$$

A.2.2 Conditional rewrite rules

Let

$$\begin{aligned} \text{lem}_1: & \quad \chi \rightarrow (\varphi_1 \quad \leftrightarrow \quad \varphi_2) \\ \text{lem}_2: & \quad \chi \rightarrow (e_1 \quad = \quad e_2) \end{aligned}$$

then

$$\frac{\Gamma \vdash lem_1 \quad \Gamma \vdash \chi \quad \Gamma \vdash_{\mathbf{c}} \varphi_2 \supset \psi}{\Gamma \vdash_{\mathbf{c}} \varphi_1 \supset \psi} \text{rwpre}\supset$$

$$\frac{\Gamma \vdash lem_1 \quad \Gamma \vdash \chi \quad \Gamma \vdash_{\mathbf{c}} \varphi_2 \subset \psi}{\Gamma \vdash_{\mathbf{c}} \varphi_1 \subset \psi} \text{rwpre}\subset$$

$$\frac{\Gamma \vdash lem_2 \quad \Gamma \vdash \chi \quad \Gamma \vdash_{\mathbf{c}} e_2 = e}{\Gamma \vdash_{\mathbf{c}} e_1 = e} \text{rwpre}=\$$

A.2.3 Weakening and strengthening rules

Let

$$\text{lem}: \quad \varphi_1 \quad \rightarrow \quad \varphi_2$$

then

$$\frac{\Gamma \vdash lem \quad \Gamma \vdash_{\mathbf{c}} \varphi_1 \supset \psi}{\Gamma \vdash_{\mathbf{c}} \varphi_2 \supset \psi} \text{strengthen}\supset \quad \frac{\Gamma \vdash lem \quad \Gamma \vdash_{\mathbf{c}} \varphi_2 \subset \psi}{\Gamma \vdash_{\mathbf{c}} \varphi_1 \subset \psi} \text{weaken}\subset$$

A.3 Congruence rules

Rules to lift sub expressions to top level.

A.3.1 Simple congruence rules

Let

$$\text{lem}: \quad e_1 = e_2 \rightarrow (f(e_1) = f(e_2))$$

then

$$\frac{\Gamma \vdash lem \quad \Gamma \vdash_{\mathbf{c}} e_1 = e_2 \quad \Gamma \vdash_{\mathbf{c}} f(e_2) \supset e}{\Gamma \vdash_{\mathbf{c}} f(e_1) \supset e} \text{cong}\supset$$

$$\frac{\Gamma \vdash lem \quad \Gamma \vdash_{\mathbf{c}} e_1 = e_2 \quad \Gamma \vdash_{\mathbf{c}} f(e_2) \subset e}{\Gamma \vdash_{\mathbf{c}} f(e_1) \subset e} \text{cong}\subset$$

$$\frac{\Gamma \vdash lem \quad \Gamma \vdash_{\mathbf{c}} e_1 = e_2 \quad \Gamma \vdash_{\mathbf{c}} f(e_2) = e}{\Gamma \vdash_{\mathbf{c}} f(e_1) = e} \text{cong}=\$$

A.3.2 Congruence rules with implication

Let

$$\text{lem: } (\varphi_1 \rightarrow \varphi_2) \rightarrow (\psi(\varphi_1) \rightarrow \psi(\varphi_2))$$

then

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma \vdash_{\mathbf{c}} \varphi_2 \supset \varphi_1 \quad \Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \supset \chi}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \chi} \text{congimp}^{\supset}$$

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma \vdash_{\mathbf{c}} \varphi_1 \subset \varphi_2 \quad \Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \subset \chi}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \subset \chi} \text{congimp}^{\subset}$$

A.3.3 Negating congruence rules

Let

$$\text{lem: } (\varphi_2 \rightarrow \varphi_1) \rightarrow (\psi(\varphi_1) \rightarrow \psi(\varphi_2))$$

then

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma \vdash_{\mathbf{c}} \varphi_2 \subset \varphi_1 \quad \Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \supset \chi}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \chi} \text{negcong}^{\supset}$$

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma \vdash_{\mathbf{c}} \varphi_1 \supset \varphi_2 \quad \Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \subset \chi}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \subset \chi} \text{negcong}^{\subset}$$

A.4 Congruence rules with additional context

Rules to lift sub expressions to top level. Additional context can be used to rewrite sub formula. Additional context formulas are normalised with the following functions.

Let

$$\begin{aligned} \text{norm}(\varphi_1 \wedge \varphi_2) &:= \text{norm}(\varphi_1) \cup \text{norm}(\varphi_2) \\ \text{norm}(\neg \varphi) &:= \{\text{neg}(\psi) \mid \psi \in \text{normneg}(\varphi)\} \\ \text{norm}(\varphi) &:= \{\varphi\} \\ \text{normneg}(\varphi_1 \vee \varphi_2) &:= \text{normneg}(\varphi_1) \cup \text{normneg}(\varphi_2) \\ \text{normneg}(\neg \varphi) &:= \{\text{neg}(\psi) \mid \psi \in \text{norm}(\varphi)\} \\ \text{normneg}(\varphi) &:= \{\varphi\} \\ \text{neg}(\neg \varphi) &:= \varphi \\ \text{neg}(\varphi) &:= \neg \varphi \end{aligned}$$

A.4.1 Simple congruence rules

Let

$$\text{lem: } (\varphi \rightarrow e_1 = e_2) \rightarrow (f(e_1) = f(e_2))$$

then

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma, \text{norm}(\varphi) \vdash_{\mathbf{c}} e_1 = e_2 \quad \Gamma \vdash_{\mathbf{c}} f(e_2) \supset e}{\Gamma \vdash_{\mathbf{c}} f(e_1) \supset e} \text{congpre}^{\supset}$$

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma, \text{norm}(\varphi) \vdash_{\mathbf{c}} e_1 = e_2 \quad \Gamma \vdash_{\mathbf{c}} f(e_2) \subset e}{\Gamma \vdash_{\mathbf{c}} f(e_1) \subset e} \text{congpre}^{\subset}$$

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma, \text{norm}(\varphi) \vdash_{\mathbf{c}} e_1 = e_2 \quad \Gamma \vdash_{\mathbf{c}} f(e_2) = e}{\Gamma \vdash_{\mathbf{c}} f(e_1) = e} \text{congpre}^{\text{=}}$$

A.4.2 Congruence rules with implication

Let

$$\text{lem: } (\varphi \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow (\psi(\varphi_1) \rightarrow \psi(\varphi_2))$$

then

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma, \text{norm}(\varphi) \vdash_{\mathbf{c}} \varphi_2 \supset \varphi_1 \quad \Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \supset \chi}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \chi} \text{congimppre}^{\supset}$$

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma, \text{norm}(\varphi) \vdash_{\mathbf{c}} \varphi_1 \subset \varphi_2 \quad \Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \subset \chi}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \subset \chi} \text{congimppre}^{\subset}$$

A.4.3 Negating congruence rules

Let

$$\text{lem: } (\varphi \rightarrow (\varphi_2 \rightarrow \varphi_1)) \rightarrow (\psi(\varphi_1) \rightarrow \psi(\varphi_2))$$

then

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma, \text{norm}(\varphi) \vdash_{\mathbf{c}} \varphi_2 \subset \varphi_1 \quad \Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \supset \chi}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \chi} \text{negcongpre}^{\supset}$$

$$\frac{\Gamma \vdash \text{lem} \quad \Gamma, \text{norm}(\varphi) \vdash_{\mathbf{c}} \varphi_1 \supset \varphi_2 \quad \Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \subset \chi}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \subset \chi} \text{negcongpre}^{\subset}$$

A.5 Congruence rules with restricted context

Rules to lift sub expressions to top level. Context must be restricted to rewrite sub expression.

A.5.1 Quantifiers

Let

$$lem: (\forall v. \varphi_1 \rightarrow \varphi_2) \rightarrow (\psi(\varphi_1) \rightarrow \psi(\varphi_2))$$

then

$$\frac{\Gamma \vdash lem \quad \Gamma[v_0/v] \vdash_{\mathbf{c}} \varphi_2 \supset \varphi_1 \quad \Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \supset \chi}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \chi} \text{allcong}^{\supset}$$

$$\frac{\Gamma \vdash lem \quad \Gamma[v_0/v] \vdash_{\mathbf{c}} \varphi_1 \subset \varphi_2 \quad \Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \subset \chi}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \subset \chi} \text{allcong}^{\subset}$$

where v_0 is fresh with respect to $\text{free}(\Gamma) \setminus \{v\} \cup \text{free}(\varphi_1, \varphi_2)$.

A.5.2 System operators

Let

$$lem: [\text{true}] (\varphi_1 \rightarrow \varphi_2) \rightarrow (\psi(\varphi_1) \rightarrow \psi(\varphi_2))$$

then

$$\frac{\Gamma \vdash lem \quad \Gamma_0 \vdash_{\mathbf{c}} \varphi_2 \supset \varphi_1 \quad \Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \supset \chi}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \chi} \text{boxcong}^{\supset}$$

$$\frac{\Gamma \vdash lem \quad \Gamma_0 \vdash_{\mathbf{c}} \varphi_1 \subset \varphi_2 \quad \Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \subset \chi}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \subset \chi} \text{boxcong}^{\subset}$$

where $\Gamma_0 := \{\varepsilon \mid \varepsilon \in \Gamma, \varepsilon \text{ is a condition}\}$. (For *conditions*, see Def. 30.)

A.5.3 System operators with next

Let

$$lem: [\text{true}] \bullet (\varphi_1 \rightarrow \varphi_2) \rightarrow (\psi(\varphi_1) \rightarrow \psi(\varphi_2))$$

then

$$\frac{\Gamma \vdash lem \quad \Gamma_0 \vdash_{\mathbf{c}} \varphi_2 \supset \varphi_1 \quad \Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \supset \chi}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \supset \chi} \text{ xboxcong}^{\supset}$$

$$\frac{\Gamma \vdash lem \quad \Gamma_0 \vdash_{\mathbf{c}} \varphi_1 \subset \varphi_2 \quad \Gamma \vdash_{\mathbf{c}} \psi(\varphi_2) \subset \chi}{\Gamma \vdash_{\mathbf{c}} \psi(\varphi_1) \subset \chi} \text{ xboxcong}^{\subset}$$

where $\Gamma_0 := \{\varepsilon_{\mathbf{s}} \mid \varepsilon_{\mathbf{s}} \in \Gamma, \varepsilon_{\mathbf{s}} \text{ is a static expression}\}$. (For *static expressions*, see Def. 30.)

Appendix B

Operators

In the following sections, the syntax, semantics, axioms, and properties of the different operators are listed. The first Section B.1 is restricted to a propositional subset of the logic, the following Section B.2 contains additional operators and rules for first order logic. Basic operators and rules for formulas in normal form as defined in Section 4.2 are given in Section B.3. The remaining operators can be found in the remaining sections.

The definitions rely on a number of auxilliary functions $\text{trm}(\rho)$, $\text{frm}(\rho, \delta)$, and $\text{frm}^c(\rho, \delta)$, the definition of these functions being as follows:

$$\begin{aligned}\text{trm}(\rho) &::= \rho^{[w,w/w',w'']} \\ \text{frm}(\rho, \text{true}) &::= \rho \\ \text{frm}(\rho, \text{false}) &::= \text{false} \\ \text{frm}(\rho, [\vec{x}]) &::= \rho^{[\vec{x}/\vec{x}']} \\ \text{frm}(\rho, \neg [\vec{x}]) &::= \rho \\ \\ \text{frm}^c(\rho, \text{true}) &::= \text{true} \\ \text{frm}^c(\rho, \text{false}) &::= \rho \\ \text{frm}^c(\rho, [\vec{x}]) &::= \rho \\ \text{frm}^c(\rho, \neg [\vec{x}]) &::= \rho^{[\vec{x}/\vec{x}']}\end{aligned}$$

The rewrite rules fall into different categories. The majority of rules can be used to automatically simplify a formula; these rules are annotated with (S). Rules (E) give several cases while rules (I) recursively unwind an operator which loops. Rules which are named *... lem* and *... rw* define congruence rules as described in Section 3.2. They are implicitly applied, if sub formulas are rewritten. Rules which are named *... dis*, *... con*, *... ex*, and *... all* are also implicitly applied to lift cases and quantifiers if necessary.

B.1 Propositional Logic

B.1.1 X (static variable)

Syntax	$X \in \mathbf{E}_s$, if $X \in \mathbf{X}_s$
Semantics	$\llbracket X \rrbracket_I := I(0)(X)$

B.1.2 w (dynamic or program variable)

Syntax	$w \in \mathbf{E}_s$, if $w \in \mathbf{Y}_s \cup \mathbf{Z}_s$
Semantics	$\llbracket x \rrbracket_I := I(0)(x)$

B.1.3 true

Syntax	$\text{true} \in \mathbf{F}$
Semantics	$I \models \text{true}$
Axioms	$\text{true gen: } \varphi \rightarrow \text{true}$

B.1.4 false

Syntax	$\text{false} \in \mathbf{F}$
Semantics	$\text{false} := \neg \text{true}$
Properties	$\text{false gen: } \text{false} \rightarrow \varphi$

B.1.5 $\neg \varphi$ (negation)

Syntax	$\neg \varphi \in \mathbf{F}$, if $\varphi \in \mathbf{F}$
Semantics	$I \models \neg \varphi$ iff $I \not\models \varphi$
Axioms	$\text{not rw: } (\varphi_1 \leftrightarrow \varphi_2) \rightarrow (\neg \varphi_1 \leftrightarrow \neg \varphi_2)$ $\text{not lem: } (\varphi_2 \rightarrow \varphi_1) \rightarrow (\neg \varphi_1 \rightarrow \neg \varphi_2)$ $\text{not dis: } \neg (\varphi_1 \vee \varphi_2) \leftrightarrow \neg \varphi_1 \wedge \neg \varphi_2$
Properties	

	<i>not con:</i>	$\neg (\varphi_1 \wedge \varphi_2) \leftrightarrow \neg \varphi_1 \vee \neg \varphi_2$
(S)	<i>not false:</i>	$\neg \text{false} \leftrightarrow \text{true}$
(S)	<i>not true:</i>	$\neg \text{true} \leftrightarrow \text{false}$
(S)	<i>not not:</i>	$\neg \neg \varphi \leftrightarrow \varphi$
(S)	<i>not connot:</i>	$\neg (\varphi_1 \wedge \neg \varphi_2) \leftrightarrow \varphi_1 \rightarrow \varphi_2$

B.1.6 $\varphi \rightarrow \psi$ (implication)

Syntax	$\varphi \rightarrow \psi \in \mathbf{F}$, if $\varphi, \psi \in \mathbf{F}$		
Semantics	$I \models \varphi \rightarrow \psi$ iff $I \models \varphi \Rightarrow I \models \psi$		
Axioms			
	<i>imp rw 1:</i>	$(\neg \psi \rightarrow (\varphi_1 \leftrightarrow \varphi_2)) \rightarrow (\varphi_1 \rightarrow \psi \leftrightarrow \varphi_2 \rightarrow \psi)$	
	<i>imp lem 1:</i>	$(\neg \psi \rightarrow (\varphi_2 \rightarrow \varphi_1)) \rightarrow ((\varphi_1 \rightarrow \psi) \rightarrow (\varphi_2 \rightarrow \psi))$	
	<i>imp rw 2:</i>	$(\psi \rightarrow (\varphi_1 \leftrightarrow \varphi_2)) \rightarrow (\psi \rightarrow \varphi_1 \leftrightarrow \psi \rightarrow \varphi_2)$	
	<i>imp lem 2:</i>	$(\psi \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow ((\psi \rightarrow \varphi_1) \rightarrow (\psi \rightarrow \varphi_2))$	
	<i>imp dis 1:</i>	$\varphi_1 \vee \varphi_2 \rightarrow \psi \leftrightarrow (\varphi_1 \rightarrow \psi) \wedge (\varphi_2 \rightarrow \psi)$	
	<i>imp con 2:</i>	$\varphi \rightarrow \psi_1 \wedge \psi_2 \leftrightarrow (\varphi \rightarrow \psi_1) \wedge (\varphi \rightarrow \psi_2)$	
(S)	<i>imp ax:</i>	$\varphi \rightarrow \varphi \leftrightarrow \text{true}$	
Properties			
(S)	<i>imp true 1:</i>	$\text{true} \rightarrow \psi \leftrightarrow \psi$	
(S)	<i>imp false 1:</i>	$\text{false} \rightarrow \psi \leftrightarrow \text{true}$	
(S)	<i>imp true 2:</i>	$\varphi \rightarrow \text{true} \leftrightarrow \text{true}$	
(S)	<i>imp false 2:</i>	$\varphi \rightarrow \text{false} \leftrightarrow \neg \varphi$	
(S)	<i>imp not 1:</i>	$\neg \varphi \rightarrow \psi \leftrightarrow \varphi \vee \psi$	
(S)	<i>imp not 2:</i>	$\psi \rightarrow \neg \varphi \leftrightarrow \neg (\psi \wedge \varphi)$	
(S)	<i>imp connot 1:</i>	$\varphi_1 \wedge \neg \varphi_2 \rightarrow \psi \leftrightarrow \varphi_1 \rightarrow \varphi_2 \vee \psi$	
(S)	<i>imp imp 2:</i>	$\psi \rightarrow \varphi_1 \rightarrow \varphi_2 \leftrightarrow \psi \wedge \varphi_1 \rightarrow \varphi_2$	
	<i>imp case:</i>	$\varphi_1 \rightarrow \varphi_2 \leftrightarrow \neg \varphi_1 \vee \varphi_2$	

B.1.7 $\varphi \wedge \psi$ (conjunction)

Syntax	$\varphi \wedge \psi \in \mathbf{F}$, if $\varphi, \psi \in \mathbf{F}$
Semantics	$\varphi \wedge \psi := \neg(\neg\varphi \vee \neg\psi)$
Properties	
<i>con rw 1:</i>	$(\psi \rightarrow (\varphi_1 \leftrightarrow \varphi_2)) \rightarrow (\varphi_1 \wedge \psi \leftrightarrow \varphi_2 \wedge \psi)$
<i>con lem 1:</i>	$(\psi \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow (\varphi_1 \wedge \psi \rightarrow \varphi_2 \wedge \psi)$
<i>con rw 2:</i>	$(\psi \rightarrow (\varphi_1 \leftrightarrow \varphi_2)) \rightarrow (\psi \wedge \varphi_1 \leftrightarrow \psi \wedge \varphi_2)$
<i>con lem 2:</i>	$(\psi \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow (\psi \wedge \varphi_1 \rightarrow \psi \wedge \varphi_2)$
<i>con dis 1:</i>	$(\varphi_1 \vee \varphi_2) \wedge \psi \leftrightarrow \varphi_1 \wedge \psi \vee \varphi_2 \wedge \psi$
<i>con dis 2:</i>	$\psi \wedge (\varphi_1 \vee \varphi_2) \leftrightarrow \psi \wedge \varphi_1 \vee \psi \wedge \varphi_2$
(S) <i>con true 1:</i>	$\text{true} \wedge \psi \leftrightarrow \psi$
(S) <i>con true 2:</i>	$\varphi \wedge \text{true} \leftrightarrow \varphi$
(S) <i>con false 1:</i>	$\text{false} \wedge \psi \leftrightarrow \text{false}$
(S) <i>con false 2:</i>	$\varphi \wedge \text{false} \leftrightarrow \text{false}$
(S) <i>con not 1:</i>	$\neg\varphi \wedge \psi^+ \leftrightarrow \psi^+ \wedge \neg\varphi$
(S) <i>con connot 2:</i>	$\psi \wedge (\varphi_1 \wedge \neg\varphi_2) \leftrightarrow (\psi \wedge \varphi_1) \wedge \neg\varphi_2$
(S) <i>con connot 1:</i>	$(\varphi_1 \wedge \neg\varphi_2) \wedge \psi^+ \leftrightarrow (\varphi_1 \wedge \psi^+) \wedge \neg\varphi_2$
	where ψ^+ is a positive formula
<i>con dpl:</i>	$\varphi \leftrightarrow \varphi \wedge \varphi$

B.1.8 $\varphi \vee \psi$ (disjunction)

Syntax	$\varphi \vee \psi \in \mathbf{F}$, if $\varphi, \psi \in \mathbf{F}$
Semantics	$\varphi \vee \psi := \neg(\neg\varphi \rightarrow \psi)$
Properties	
<i>dis rw 1:</i>	$(\neg\psi \rightarrow (\varphi_1 \leftrightarrow \varphi_2)) \rightarrow (\varphi_1 \vee \psi \leftrightarrow \varphi_2 \vee \psi)$
<i>dis lem 1:</i>	$(\neg\psi \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow (\varphi_1 \vee \psi \rightarrow \varphi_2 \vee \psi)$
<i>dis rw 2:</i>	$(\neg\psi \rightarrow (\varphi_1 \leftrightarrow \varphi_2)) \rightarrow (\psi \vee \varphi_1 \leftrightarrow \psi \vee \varphi_2)$

	<i>dis lem 2:</i>	$(\neg \psi \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow (\psi \vee \varphi_1 \rightarrow \psi \vee \varphi_2)$
	<i>dis con 1:</i>	$\varphi_1 \wedge \varphi_2 \vee \psi \leftrightarrow (\varphi_1 \vee \psi) \wedge (\varphi_2 \vee \psi)$
	<i>dis con 2:</i>	$\varphi \vee \psi_1 \wedge \psi_2 \leftrightarrow (\varphi \vee \psi_1) \wedge (\varphi \vee \psi_2)$
(S)	<i>dis false 1:</i>	$\text{false} \vee \psi \leftrightarrow \psi$
(S)	<i>dis false 2:</i>	$\varphi \vee \text{false} \leftrightarrow \varphi$
(S)	<i>dis true 1:</i>	$\text{true} \vee \psi \leftrightarrow \text{true}$
(S)	<i>dis true 2:</i>	$\varphi \vee \text{true} \leftrightarrow \text{true}$
(S)	<i>dis not 1:</i>	$\neg \varphi \vee \psi \leftrightarrow \varphi \rightarrow \psi$
(S)	<i>dis not 2:</i>	$\psi \vee \neg \varphi \leftrightarrow \varphi \rightarrow \psi$
(S)	<i>dis imp 1:</i>	$(\varphi_1 \rightarrow \varphi_2) \vee \psi \leftrightarrow \varphi_1 \rightarrow \varphi_2 \vee \psi$
(S)	<i>dis imp 2:</i>	$\psi \vee (\varphi_1 \rightarrow \varphi_2) \leftrightarrow \varphi_1 \rightarrow \psi \vee \varphi_2$

B.1.9 $\varphi \leftrightarrow \psi$ (equivalence)

Syntax	$\varphi \leftrightarrow \psi \in \mathbf{F}$, if $\varphi, \psi \in \mathbf{F}$		
Semantics	$\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$		
Properties			
	<i>equiv rw 1:</i>	$(\varphi_1 \leftrightarrow \varphi_2) \rightarrow ((\varphi_1 \leftrightarrow \psi) \leftrightarrow (\varphi_2 \leftrightarrow \psi))$	
	<i>equiv rw 2:</i>	$(\varphi_1 \leftrightarrow \varphi_2) \rightarrow ((\psi \leftrightarrow \varphi_1) \leftrightarrow (\psi \leftrightarrow \varphi_2))$	
(S)	<i>equiv true 1:</i>	$(\text{true} \leftrightarrow \psi) \leftrightarrow \psi$	
(S)	<i>equiv true 2:</i>	$(\varphi \leftrightarrow \text{true}) \leftrightarrow \varphi$	
(S)	<i>equiv false 1:</i>	$(\text{false} \leftrightarrow \psi) \leftrightarrow \neg \psi$	
(S)	<i>equiv false 2:</i>	$(\varphi \leftrightarrow \text{false}) \leftrightarrow \neg \varphi$	
	<i>equiv case:</i>	$(\varphi_1 \leftrightarrow \varphi_2) \leftrightarrow \varphi_1 \wedge \varphi_2 \vee \neg \varphi_1 \wedge \neg \varphi_2$	
	<i>equiv case cnf:</i>	$(\varphi_1 \leftrightarrow \varphi_2) \leftrightarrow (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$	

B.2 First Order Logic

B.2.1 w' (primed variable)

Syntax	$w' \in \mathbf{E}_s$, if $w \in \mathbf{Y}_s \cup \mathbf{Z}_s$
Semantics	

$\llbracket w' \rrbracket_I$	iff	$\begin{cases} I(0)'(w), & \text{if } I > 0 \\ I(0)(w), & \text{otherwise} \end{cases}$
------------------------------	-----	---

B.2.2 w'' (double primed variable)

Syntax	$w'' \in \mathbf{E}_s$, if $w \in \mathbf{Y}_s \cup \mathbf{Z}_s$
Semantics	$\llbracket w'' \rrbracket_I \quad \text{iff} \quad \begin{cases} I(1)(w), & \text{if } I > 0 \\ I(0)(w), & \text{otherwise} \end{cases}$
Properties	<i>dprm snx</i> : $w'' = X \quad \leftrightarrow \quad \circ w = X$

B.2.3 $f(e_1, \dots, e_n)$ (function application)

Syntax	$f(e_1, \dots, e_n) \in \mathbf{E}_s$, if $f \in \mathbf{OP}_{s_1, \dots, s_n}^s, e_i \in \mathbf{E}_{s_i}$
Semantics	$\llbracket f(e_1, \dots, e_n) \rrbracket_{A,I} := f_A(\llbracket e_1 \rrbracket_{A,I}, \dots, \llbracket e_n \rrbracket_{A,I})$
Axioms	<i>app lem i</i> : $e_i = e \rightarrow (f(\dots, e_i, \dots) = f(\dots, e, \dots))$

B.2.4 $e_1 = e_2$ (equation)

Syntax	$e_1 = e_2 \in \mathbf{F}$, if $e_1, e_2 \in \mathbf{E}_s$
Semantics	$I \models (e_1 = e_2)$ iff $\llbracket e_1 \rrbracket_I = \llbracket e_2 \rrbracket_I$
Axioms	<i>eq lem 1</i> : $e_1 = e_2 \rightarrow (e_1 = e \leftrightarrow e_2 = e)$ <i>eq lem 2</i> : $e_1 = e_2 \rightarrow (e = e_1 \leftrightarrow e = e_2)$ (S) <i>eq refl</i> : $e = e \leftrightarrow \text{true}$

B.2.5 $\exists v. \varphi$ (existential quantification)

Syntax	$\exists v. \varphi \in \mathbf{F}$, if $v \in \mathbf{X} \cup \mathbf{Y}, \varphi \in \mathbf{F}$
Semantics	$I \models \exists v. \varphi$ iff there exists I_0 with $I_0 =_v I$ and $I_0 \models \varphi$
Axioms	<i>ex rw</i> : $(\forall v. \varphi_1 \leftrightarrow \varphi_2) \rightarrow ((\exists v. \varphi_1) \leftrightarrow (\exists v. \varphi_2))$ <i>ex lem</i> : $(\forall v. \varphi_1 \rightarrow \varphi_2) \rightarrow ((\exists v. \varphi_1) \rightarrow (\exists v. \varphi_2))$

$\begin{aligned} \text{ex dis:} & & (\exists v. \varphi_1 \vee \varphi_2) & \leftrightarrow & (\exists v. \varphi_1) \vee (\exists v. \varphi_2) \\ \text{ex inst:} & & (\exists v. \varphi) & \leftrightarrow & \varphi[e/x] \vee (\exists v. \varphi) \end{aligned}$ <p style="text-align: center; margin: 0;">where $e \in \mathbf{E}_s$, if $v \in \mathbf{X}_s \cup \mathbf{Y}_s$</p>
<p>Properties</p> $\text{(S) ex elim: } (\exists v. \varphi) \leftrightarrow \varphi$ <p style="text-align: center; margin: 0;">where $v \notin \text{free}(\varphi)$</p>

B.2.6 $\forall v. \varphi$ (universal quantification)

Syntax	$\forall v. \varphi \in \mathbf{F}$, if $v \in \mathbf{X}$, $\varphi \in \mathbf{F}$
Semantics	$\forall v. \varphi := \neg \exists v. \neg \varphi$
Properties	
<i>all rw:</i>	$(\forall v. \varphi_1 \leftrightarrow \varphi_2) \rightarrow ((\forall v. \varphi_1) \leftrightarrow (\forall v. \varphi_2))$
<i>all lem:</i>	$(\forall v. \varphi_1 \rightarrow \varphi_2) \rightarrow ((\forall v. \varphi_1) \rightarrow (\forall v. \varphi_2))$
<i>all con:</i>	$(\forall v. \varphi_1 \wedge \varphi_2) \leftrightarrow (\forall v. \varphi_1) \wedge (\forall v. \varphi_2)$
<i>all inst:</i>	$(\forall v. \varphi) \leftrightarrow \varphi[e/x] \wedge (\forall v. \varphi)$
where $e \in \mathbf{E}_s$, if $v \in \mathbf{X}_s \cup \mathbf{Y}_s$	
(S) <i>all elim:</i>	$(\forall v. \varphi) \leftrightarrow \varphi$
where $v \notin \text{free}(\varphi)$	

B.2.7 Negation of quantifiers

Axioms	$\text{not ex: } \neg (\exists v. \varphi) \leftrightarrow \forall v. \neg \varphi$
Properties	$\text{not all: } \neg (\forall v. \varphi) \leftrightarrow \exists v. \neg \varphi$

B.2.8 Implication of quantifiers

Axioms	
<i>imp ex 1:</i>	$(\exists v. \varphi) \rightarrow \psi \leftrightarrow \forall v_0. \varphi[v_0/v] \rightarrow \psi$
<i>imp all 2:</i>	$\psi \vee (\forall v. \varphi) \leftrightarrow \forall v_0. \psi \rightarrow \varphi[v_0/v]$
v_0 fresh with respect to $(\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(\psi)$	

B.2.9 Conjunction of quantifiers

Properties	
<i>con ex 1:</i>	$(\exists v. \varphi) \wedge \psi \leftrightarrow \exists v_0. \varphi^{[v_0/v]} \wedge \psi$
<i>con ex 2:</i>	$\psi \wedge (\exists v. \varphi) \leftrightarrow \exists v_0. \psi \wedge \varphi^{[v_0/v]}$
	v_0 fresh with respect to $(\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(\psi)$

B.2.10 Disjunction of quantifiers

Properties	
<i>dis all 1:</i>	$(\forall v. \varphi) \vee \psi \leftrightarrow \forall v_0. \varphi^{[v_0/v]} \vee \psi$
<i>dis all 2:</i>	$\psi \vee (\forall v. \varphi) \leftrightarrow \forall v_0. \psi \vee \varphi^{[v_0/v]}$
	v_0 fresh with respect to $(\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(\psi)$

B.3 Transitions

B.3.1 Transition predicate

Axioms	
<i>pl:</i>	$\rho \leftrightarrow \rho \wedge \text{true} \wedge \text{true} \wedge \bullet \text{true}$
Properties	
<i>pl cnf:</i>	$\rho \leftrightarrow (\rho \vee \text{false} \vee \text{false} \vee \circ \text{false})$

B.3.2 $[\vec{x}]$ (frame assumption)

Syntax	$[\vec{x}] \in \mathbf{F}$, if $\vec{x} \subset \mathbf{Z}$, \vec{x} finite
Semantics	$I \models [\vec{x}]$ iff $I(0)(x') = I(0)(x)$ for all $x \notin \vec{x}$
Axioms	
<i>frm:</i>	$[\vec{x}] \leftrightarrow \text{true} \wedge [\vec{x}] \wedge \text{true} \wedge \bullet \text{true}$
<i>frm elim:</i>	$[\vec{x}_0] \rightarrow ([\vec{x}] \leftrightarrow \bigwedge \{x' = x \mid x \in \vec{x}_0 \setminus \vec{x}\})$
(S) <i>frm con pp:</i>	$[\vec{x}_1] \wedge [\vec{x}_2] \leftrightarrow [\vec{x}_1 \cap \vec{x}_2]$
(S) <i>frm con pn:</i>	$[\vec{x}_1] \wedge \neg [\vec{x}_2] \leftrightarrow (\bigvee_{x \in \vec{x}_1 \setminus \vec{x}_2} x' \neq x) \wedge [\vec{x}_1]$
(S) <i>frm con np:</i>	$\neg [\vec{x}_1] \wedge [\vec{x}_2] \leftrightarrow (\bigvee_{x \in \vec{x}_2 \setminus \vec{x}_1} x' \neq x) \wedge [\vec{x}_2]$

(S) <i>frm con nn:</i>	$\neg [\vec{x}_1] \wedge \neg [\vec{x}_2]$	\leftrightarrow	$(\bigvee_{x \in \vec{x}_1 \setminus \vec{x}_2} x' \neq x)$ $\wedge (\bigvee_{x \in \vec{x}_2 \setminus \vec{x}_1} x' \neq x)$ $\vee \neg [\vec{x}_1 \cup \vec{x}_2]$
(S) <i>frm dis pp:</i>	$[\vec{x}_1] \vee [\vec{x}_2]$	\leftrightarrow	$(\bigwedge_{x \in \vec{x}_1 \setminus \vec{x}_2} x' = x$ $\vee \bigwedge_{x \in \vec{x}_2 \setminus \vec{x}_1} x' = x)$ $\wedge [\vec{x}_1 \cup \vec{x}_2]$
(S) <i>frm dis pn:</i>	$[\vec{x}_1] \vee \neg [\vec{x}_2]$	\leftrightarrow	$\bigwedge_{x \in \vec{x}_2 \setminus \vec{x}_1} x' = x \vee \neg [\vec{x}_2]$
(S) <i>frm dis np:</i>	$\neg [\vec{x}_1] \vee [\vec{x}_2]$	\leftrightarrow	$\bigwedge_{x \in \vec{x}_1 \setminus \vec{x}_2} x' = x \vee \neg [\vec{x}_1]$
(S) <i>frm dis nn:</i>	$\neg [\vec{x}_1] \vee \neg [\vec{x}_2]$	\leftrightarrow	$\neg [\vec{x}_1 \cap \vec{x}_2]$

B.3.3 blk

Syntax	blocked $\in \mathbf{F}$
Semantics	blocked $:\Leftrightarrow \neg \text{blk}' = \text{blk}$
Axioms	<i>blk:</i> blocked $\leftrightarrow \text{true} \wedge \text{true} \wedge \mathbf{blocked} \wedge \circ \text{true}$

B.3.4 last (last step)

Syntax	last $\in \mathbf{F}$
Semantics	last $:\equiv \bullet \text{false}$
Properties	<i>lst:</i> last $\leftrightarrow \text{true} \wedge \mathbf{last}$

B.3.5 $\circ \varphi$ (strong next)

Syntax	$\circ \varphi \in \mathbf{F}$, if $\varphi \in \mathbf{F}$
Semantics	$I \models \circ \varphi$ iff $0 < I $ and $I _1 \models \varphi$
Axioms	<i>snx:</i> $\circ \varphi \leftrightarrow \text{true} \wedge \text{true} \wedge \text{true} \wedge \circ \varphi$
Properties	<i>snx cnf:</i> $\circ \varphi \leftrightarrow \text{false} \vee \text{false} \vee \text{false} \vee \circ \varphi$

B.3.6 $\bullet \varphi$ (weak next)

Syntax	$\bullet \varphi \in \mathbf{F}$, if $\varphi \in \mathbf{F}$
Semantics	$\bullet \varphi := \neg \circ \neg \varphi$
Properties	<i>wax</i> : $\bullet \varphi \leftrightarrow \text{true} \wedge \mathbf{last} \vee \text{true} \wedge \text{true} \wedge \text{true} \wedge \circ \varphi$

B.3.7 Negation of transition

Axioms	<p>(S) <i>not lst</i>: $\neg(\rho \wedge \mathbf{last}) \leftrightarrow \neg \rho \vee \neg \mathbf{last}$</p> <p>(S) <i>not stp</i>: $\neg(\rho \wedge \delta \wedge \beta \wedge \circ \varphi) \leftrightarrow \neg \rho \vee \neg \delta \vee \neg \beta \vee \bullet \neg \varphi$</p>
Properties	<p>(S) <i>not lst dnf</i>: $\neg(\rho \vee \neg \mathbf{last}) \leftrightarrow (\neg \rho \wedge \mathbf{last})$</p> <p>(S) <i>not stp dnf</i>: $\neg(\rho \vee \delta \vee \beta \vee \bullet \varphi) \leftrightarrow (\neg \rho \wedge \neg \delta \wedge \neg \beta \wedge \circ \neg \varphi)$</p> <p>(S) <i>not wstp dnf</i>: $\neg(\rho \vee \delta \vee \beta \vee \circ \varphi) \leftrightarrow (\neg \rho \wedge \neg \delta \wedge \neg \beta \wedge \bullet \neg \varphi)$</p>

B.3.8 Implication of transitions

Axioms	<p>(S) <i>imp ll</i>: $\rho_1 \wedge \mathbf{last} \rightarrow \rho_2 \vee \neg \mathbf{last} \leftrightarrow (\rho_1 \rightarrow \rho_2) \vee \neg \mathbf{last}$</p> <p>(S) <i>imp ls</i>: $\rho_1 \wedge \mathbf{last} \rightarrow \rho_2 \vee \delta_2 \vee \beta_2 \vee \bullet \varphi_2 \leftrightarrow \text{true}$</p> <p>(S) <i>imp sl</i>: $\rho_1 \wedge \delta_1 \wedge \beta_1 \wedge \circ \varphi_1 \rightarrow \rho_2 \vee \neg \mathbf{last} \leftrightarrow \text{true}$</p> <p>(S) <i>imp ss</i>: $\rho_1 \wedge \delta_1 \wedge \beta_1 \wedge \circ \varphi_1 \rightarrow \rho_2 \vee \delta_2 \vee \beta_2 \vee \bullet \varphi_2 \leftrightarrow (\rho_1 \rightarrow \rho_2) \vee (\neg \delta_1 \vee \delta_2) \vee (\neg \beta_1 \vee \beta_2) \vee \bullet (\varphi_1 \rightarrow \varphi_2)$</p>
Properties	<p>(S) <i>imp sw</i>: $\rho_1 \wedge \delta_1 \wedge \beta_1 \wedge \circ \varphi_1 \rightarrow \rho_2 \vee \delta_2 \vee \beta_2 \vee \circ \varphi_2 \leftrightarrow (\rho_1 \rightarrow \rho_2) \vee (\neg \delta_1 \vee \delta_2) \vee (\neg \beta_1 \vee \beta_2) \vee \bullet (\varphi_1 \rightarrow \varphi_2)$</p> <p>(S) <i>imp ww</i>: $\rho_1 \wedge \delta_1 \wedge \beta_1 \wedge \bullet \varphi_1 \rightarrow \rho_2 \vee \delta_2 \vee \beta_2 \vee \circ \varphi_2 \leftrightarrow (\rho_1 \rightarrow \rho_2) \vee (\neg \delta_1 \vee \delta_2) \vee (\neg \beta_1 \vee \beta_2) \vee \circ (\varphi_1 \rightarrow \varphi_2)$</p>

B.3.9 Conjunction of transitions

Properties	
-------------------	--

(S)	<i>con ll:</i>	$(\rho_1 \wedge \mathbf{last}) \wedge (\rho_2 \wedge \mathbf{last})$ $\leftrightarrow ((\rho_1 \wedge \rho_2) \wedge \mathbf{last})$
(S)	<i>con ls:</i>	$(\rho_1 \wedge \mathbf{last}) \wedge (\rho_2 \wedge \delta_2 \wedge \beta_2 \wedge \circ \varphi_2) \leftrightarrow \mathbf{false}$
(S)	<i>con sl:</i>	$(\rho_1 \wedge \delta_1 \wedge \beta_1 \wedge \circ \varphi_1) \wedge (\rho_2 \wedge \mathbf{last}) \leftrightarrow \mathbf{false}$
(S)	<i>con ss:</i>	$(\rho_1 \wedge \delta_1 \wedge \beta_1 \wedge \circ \varphi_1) \wedge (\rho_2 \wedge \delta_2 \wedge \beta_2 \wedge \circ \varphi_2)$ $\leftrightarrow ((\rho_1 \wedge \rho_2) \wedge (\delta_1 \wedge \delta_2) \wedge (\beta_1 \wedge \beta_2) \wedge \circ (\varphi_1 \wedge \varphi_2))$
(S)	<i>con ww:</i>	$(\rho_1 \wedge \delta_1 \wedge \beta_1 \wedge \bullet \varphi_1) \wedge (\rho_2 \wedge \delta_2 \wedge \beta_2 \wedge \bullet \varphi_2)$ $\leftrightarrow (\rho_1 \wedge \rho_2 \wedge \delta_1 \wedge \delta_2 \wedge \beta_1 \wedge \beta_2 \wedge \bullet (\varphi_1 \wedge \varphi_2))$
(S)	<i>con seq:</i>	$(\rho_1 \vee \delta \vee \beta \vee \varphi) \wedge (\rho_2 \vee \delta \vee \beta \vee \varphi)$ $\leftrightarrow (\rho_1 \wedge \rho_2 \vee \delta \vee \beta \vee \varphi)$

B.3.10 Disjunction of transitions

Properties		
(S)	<i>dis ll:</i>	$(\rho_1 \vee \neg \mathbf{last}) \vee (\rho_2 \vee \neg \mathbf{last})$ $\leftrightarrow (\rho_1 \vee \rho_2 \vee \neg \mathbf{last})$
(S)	<i>dis ss:</i>	$(\rho_1 \vee \delta_1 \vee \beta_1 \vee \bullet \varphi_1) \vee (\rho_2 \vee \delta_2 \vee \beta_2 \vee \bullet \varphi_2)$ $\leftrightarrow (\rho_1 \vee \rho_2 \vee \delta_1 \vee \delta_2 \vee \beta_1 \vee \beta_2 \vee \bullet (\varphi_1 \vee \varphi_2))$
(S)	<i>dis ww:</i>	$(\rho_1 \vee \delta_1 \vee \beta_1 \vee \circ \varphi_1) \vee (\rho_2 \vee \delta_2 \vee \beta_2 \vee \circ \varphi_2)$ $\leftrightarrow (\rho_1 \vee \rho_2 \vee \delta_1 \vee \delta_2 \vee \beta_1 \vee \beta_2 \vee \circ (\varphi_1 \vee \varphi_2))$
(S)	<i>dis seq:</i>	$(\rho_1 \wedge \delta_1 \wedge \beta \wedge \varphi) \vee (\rho_2 \wedge \delta_2 \wedge \beta \wedge \varphi)$ $\leftrightarrow ((\rho_1 \wedge \bigwedge_{x \in \delta_2 \setminus \delta_1} x' = x \vee \rho_2 \wedge \bigwedge_{x \in \delta_1 \setminus \delta_2} x' = x) \wedge \delta_1 \cup \delta_2 \wedge \beta \wedge \varphi)$

B.3.11 Existential quantification of transitions

Axioms		
(S)	<i>ex lst:</i>	$(\exists v. \rho \wedge \mathbf{last}) \leftrightarrow (\exists v. \rho) \wedge \mathbf{last}$
Properties		
(S)	<i>ex stp 1:</i>	$(\exists v. \rho \wedge \delta \wedge \beta \wedge \circ \varphi) \leftrightarrow (\exists v. \rho) \wedge \delta \wedge \beta \wedge \circ \varphi$ where $v \notin \text{free}(\varphi)$
(S)	<i>ex stp 2:</i>	$(\exists v. \rho \wedge \delta \wedge \beta \wedge \circ \varphi) \leftrightarrow \rho \wedge \delta \wedge \beta \wedge \circ (\exists v. \varphi)$ where $v \notin \text{free}(\rho)$

B.3.12 Universal quantification of transitions

Properties		
-------------------	--	--

$$(S) \quad \text{all lst: } (\forall v. (\rho \wedge \mathbf{last})) \leftrightarrow ((\forall x. \rho) \wedge \mathbf{last})$$

B.3.13 Normalisation

Properties

<i>dnf swp:</i>	$\tau \wedge \circ \varphi \vee \rho \wedge \mathbf{last}$	
	$\leftrightarrow \rho \wedge \mathbf{last} \vee \tau \wedge \circ \varphi$	
<i>dnf lsts:</i>	$\rho_1 \wedge \mathbf{last} \vee \rho_2 \wedge \mathbf{last}$	
	$\leftrightarrow (\rho_1 \vee \rho_2) \wedge \mathbf{last}$	
<i>cnf swp:</i>	$(\tau \vee \bullet \varphi) \wedge (\rho \vee \neg \mathbf{last})$	
	$\leftrightarrow (\rho \vee \neg \mathbf{last}) \wedge (\tau \vee \circ \varphi)$	
<i>cnf lsts:</i>	$(\rho_1 \vee \neg \mathbf{last}) \wedge (\rho_2 \vee \neg \mathbf{last})$	
	$\leftrightarrow \rho_1 \wedge \rho_2 \vee \neg \mathbf{last}$	
<i>dnf frm tau:</i>	$(\rho \wedge \rho_0 \wedge \beta \wedge \varphi)$	
	$\leftrightarrow ((\rho \wedge \rho_0) \wedge \mathbf{true} \wedge \beta \wedge \varphi)$	
<i>dnf frm tau 2:</i>	$(\rho \wedge (\rho_0 \wedge \psi) \wedge \beta \wedge \varphi)$	
	$\leftrightarrow ((\rho \wedge \rho_0) \wedge \psi \wedge \beta \wedge \varphi)$	
<i>cnf frm tau:</i>	$\rho_1 \vee (\rho_2 \vee \delta) \vee \beta \vee \varphi$	
	$\leftrightarrow (\rho_1 \vee \rho_2) \vee \delta \vee \beta \vee \varphi$	
<i>cnf frm tau:</i>	$(\rho_1 \vee \rho_2 \vee \beta \vee \varphi)$	
	$\leftrightarrow (\rho_1 \vee \rho_2) \vee \mathbf{true} \vee \beta \vee \varphi$	
<i>dnf dis 2:</i>	$(\rho \wedge (\psi_1 \vee \psi_2) \wedge \beta \wedge \varphi)$	$\leftrightarrow (\rho \wedge \psi_1 \wedge \beta \wedge \varphi)$
		$\vee (\rho \wedge \psi_2 \wedge \beta \wedge \varphi)$
<i>dnf dis 4:</i>	$(\rho \wedge \delta \wedge \beta \wedge (\varphi_1 \vee \varphi_2))$	$\leftrightarrow (\rho \wedge \delta \wedge \beta \wedge \varphi_1)$
		$\vee (\rho \wedge \delta \wedge \beta \wedge \varphi_2)$
<i>dnf lst cnf:</i>	$(\rho \wedge \mathbf{last})$	
	$\leftrightarrow (\rho \vee \neg \mathbf{last})$	
	$\wedge (\mathbf{false} \vee \mathbf{false} \vee \mathbf{false} \vee \bullet \mathbf{false})$	
<i>dnf stp cnf:</i>	$(\rho \wedge \delta \wedge \beta \wedge \circ \varphi)$	
	$\leftrightarrow (\rho_0 \vee \mathbf{false} \vee \mathbf{false} \vee \bullet \mathbf{false})$	
	$\wedge (\delta \vee \mathbf{false} \vee \mathbf{false} \vee \bullet \mathbf{false})$	
	$\wedge (\beta \vee \mathbf{false} \vee \mathbf{false} \vee \bullet \mathbf{false})$	
	$\wedge (\mathbf{false} \vee \mathbf{false} \vee \mathbf{false} \vee \circ \varphi)$	
<i>cnf con 2:</i>	$(\rho \vee (\psi_1 \wedge \psi_2) \vee \beta \vee \varphi)$	$\leftrightarrow (\rho \vee \psi_1 \vee \beta \vee \varphi)$
		$\wedge (\rho \vee \psi_2 \vee \beta \vee \varphi)$
<i>cnf con 4:</i>	$(\rho \vee \delta \vee \beta \vee (\varphi_1 \wedge \varphi_2))$	$\leftrightarrow (\rho \vee \delta \vee \beta \vee \varphi_1)$
		$\wedge (\rho \vee \delta \vee \beta \vee \varphi_2)$

B.4 System Operators

B.4.1 $\langle \varphi \rangle \psi$ (diamond)

Syntax	$\langle \varphi \rangle \psi \in \mathbf{F}$, if $\varphi, \psi \in \mathbf{F}$
Semantics	$I \models \langle \varphi \rangle \psi \quad \text{iff} \quad \text{there exists } I_0$ $\text{with } I_0(0) = I(0)$ $\text{and } I_0 \models \varphi \text{ and } I_0 \models \psi$
Axioms	<p><i>dia norm:</i> $\langle \varphi \rangle \psi \leftrightarrow \langle \text{true} \rangle (\varphi \wedge \psi)$</p> <p><i>dia rw:</i> $[\text{true}] (\varphi_1 \leftrightarrow \varphi_2) \rightarrow (\langle \text{true} \rangle \varphi_1 \leftrightarrow \langle \text{true} \rangle \varphi_2)$</p> <p><i>dia lem:</i> $[\text{true}] (\varphi_1 \rightarrow \varphi_2) \rightarrow (\langle \text{true} \rangle \varphi_1 \rightarrow \langle \text{true} \rangle \varphi_2)$</p> <p><i>dia dis:</i> $\langle \text{true} \rangle (\varphi_1 \vee \varphi_2)$ $\leftrightarrow \langle \text{true} \rangle \varphi_1 \vee \langle \text{true} \rangle \varphi_2$</p> <p><i>dia ex:</i> $\langle \text{true} \rangle \exists v. \varphi \leftrightarrow \exists v. \langle \text{true} \rangle \varphi$</p> <p>(S) <i>dia lst:</i> $\langle \text{true} \rangle (\rho \wedge \mathbf{last})$ $\leftrightarrow \rho^{[w,w/w',w'']}$</p> <p>(S) <i>dia stp:</i> $\langle \text{true} \rangle (\rho \wedge \delta \wedge \beta \wedge \circ \varphi)$ $\leftrightarrow \exists X_2. (\exists X_1. \text{frm}(\rho, \delta)^{[X_1, X_2/w', w'']})$ $\wedge \langle \text{true} \rangle \circ (\wedge (w = X_2) \wedge \varphi)$</p>

B.4.2 $\langle \text{true} \rangle \circ \varphi$ (diamond next)

Axioms	<p><i>xdia rw:</i> $[\text{true}] (\varphi_1 \leftrightarrow \varphi_2) \rightarrow (\langle \text{true} \rangle \circ \varphi_1 \leftrightarrow \langle \text{true} \rangle \circ \varphi_2)$</p> <p><i>xdia lem:</i> $[\text{true}] (\varphi_1 \rightarrow \varphi_2) \rightarrow (\langle \text{true} \rangle \circ \varphi_1 \rightarrow \langle \text{true} \rangle \circ \varphi_2)$</p> <p><i>xdia dis:</i> $\langle \text{true} \rangle \circ (\varphi_1 \vee \varphi_2)$ $\leftrightarrow \langle \text{true} \rangle \circ \varphi_1 \vee \langle \text{true} \rangle \circ \varphi_2$</p> <p><i>xdia ex:</i> $\langle \text{true} \rangle \circ \exists v. \varphi \leftrightarrow \exists v. \langle \text{true} \rangle \circ \varphi$</p> <p>(S) <i>xdia lst:</i> $\langle \text{true} \rangle \circ (\rho \wedge \mathbf{last})$ $\leftrightarrow \exists X. \rho^{[X, X, X/w, w', w'']}$</p>
---------------	---

$$\begin{aligned}
\text{(S) } xdia \text{ stp: } & \langle \text{true} \rangle \circ (\rho \wedge \delta \wedge \beta \wedge \circ \varphi) \\
& \leftrightarrow \exists X_2. \quad (\exists X_0, X_1. \text{frm}(\rho, \delta)[^{X_0, X_1, X_2}/_{w, w', w''}]) \\
& \quad \wedge \langle \text{true} \rangle \circ (\wedge (w = X_2) \wedge \varphi)
\end{aligned}$$

B.4.3 $[\varphi] \psi$ (box)

Semantics	$[\varphi] \psi \equiv \neg \langle \varphi \rangle \neg \psi$
Properties	
<i>box norm:</i>	$[\varphi] \psi \leftrightarrow [\text{true}] (\varphi \rightarrow \psi)$
<i>box rw:</i>	$[\text{true}] (\varphi_1 \leftrightarrow \varphi_2) \rightarrow ([\text{true}] \varphi_1 \leftrightarrow [\text{true}] \varphi_2)$
<i>box lem:</i>	$[\text{true}] (\varphi_1 \rightarrow \varphi_2) \rightarrow ([\text{true}] \varphi_1 \rightarrow [\text{true}] \varphi_2)$
<i>box con:</i>	$[\text{true}] (\varphi_1 \wedge \varphi_2) \leftrightarrow [\text{true}] \varphi_1 \wedge [\text{true}] \varphi_2$
<i>box all:</i>	$[\text{true}] (\forall v. \varphi) \leftrightarrow \forall v. [\text{true}] \varphi$
(S) <i>box lst:</i>	$[\text{true}] (\rho \vee \neg \mathbf{last}) \leftrightarrow \text{trm}(\rho)$
(S) <i>box stp:</i>	$[\text{true}] (\rho \vee \delta \vee \beta \vee \bullet \varphi) \leftrightarrow \forall X_2. \quad (\forall X_1. \text{frm}^c(\rho, \delta)[^{X_1, X_2}/_{v', v''}]) \vee [\text{true}] \bullet (X_2 = w \rightarrow \varphi)$

B.4.4 $[\text{true}] \bullet \psi$ (box next)

Properties	
<i>xbox rw:</i>	$[\text{true}] \bullet (\varphi_1 \leftrightarrow \varphi_2) \rightarrow ([\text{true}] \bullet \varphi_1 \leftrightarrow [\text{true}] \bullet \varphi_2)$
<i>xbox lem:</i>	$[\text{true}] \bullet (\varphi_1 \rightarrow \varphi_2) \rightarrow ([\text{true}] \bullet \varphi_1 \rightarrow [\text{true}] \bullet \varphi_2)$
<i>xbox con:</i>	$[\text{true}] \bullet (\varphi_1 \wedge \varphi_2) \leftrightarrow [\text{true}] \bullet \varphi_1 \wedge [\text{true}] \bullet \varphi_2$
<i>xbox all:</i>	$[\text{true}] \bullet (\forall v. \varphi) \leftrightarrow \forall v. [\text{true}] \bullet \varphi$
(S) <i>xbox lst:</i>	$[\text{true}] \bullet (\rho \vee \neg \mathbf{last}) \leftrightarrow \forall X. \rho[^{X, X, X}/_{w, w', w''}]$
(S) <i>xbox stp:</i>	$[\text{true}] \bullet (\rho \vee \delta \vee \beta \vee \bullet \varphi) \leftrightarrow \forall X_2. \quad (\forall X_0, X_1. \text{frm}^c(\rho, \delta)[^{X_0, X_1, X_2}/_{v, v', v''}]) \vee [\text{true}] \bullet (X_2 = w \rightarrow \varphi)$

B.5 ITL Operators

B.5.1 $\varphi; \psi$ (chop)

Syntax	$\varphi; \psi \in \mathbf{F}$, if $\varphi, \psi \in \mathbf{F}$
Semantics	$I \models \varphi; \psi \quad \text{iff} \quad \begin{array}{l} \text{there exists } n \leq I \text{ with } I^n \models \varphi \text{ and } I _n \models \psi \\ \text{or } I = \infty \text{ and } I \models \varphi \end{array}$
Axioms	$\begin{array}{l} \text{chp rw:} \quad [\text{true}] (\varphi_1 \leftrightarrow \varphi_2) \rightarrow (\varphi_1; \psi \leftrightarrow \varphi_2; \psi) \\ \text{chp lem:} \quad [\text{true}] (\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1; \psi \rightarrow \varphi_2; \psi) \\ \text{chp dis:} \quad (\varphi_1 \vee \varphi_2); \psi \leftrightarrow \varphi_1; \psi \vee \varphi_2; \psi \\ \text{chp ex:} \quad (\exists v. \varphi); \psi \leftrightarrow \exists v_0. \varphi[v_0/v]; \psi \\ \quad \quad \quad v_0 \text{ fresh with respect to } (\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(\psi) \\ \text{(S) chp lst:} \quad (\rho \wedge \mathbf{last}); \psi \leftrightarrow \rho[w, w'/w'', w''] \wedge \psi \\ \text{(S) chp stp:} \quad (\tau^c \wedge \circ \varphi); \psi \leftrightarrow (\tau^c \wedge \circ (\varphi; \psi)) \\ \text{chp live:} \quad (\exists l. \diamond l \wedge \varphi); \psi \rightarrow \exists l_0. \diamond l_0 \wedge \varphi[l_0/l]; \psi \\ \quad \quad \quad l_0 \text{ fresh with respect to } (\text{free}(\varphi) \setminus \{l\}) \cup \text{free}(\psi) \end{array}$
Properties	$\begin{array}{l} \text{chp lem 2:} \quad \square (\varphi_1 \rightarrow \varphi_2) \rightarrow (\psi; \varphi_1 \rightarrow \psi; \varphi_2) \\ \text{(S) chp true true:} \quad \text{true; true} \leftrightarrow \text{true} \\ \text{(S) chp false:} \quad \text{false; } \psi \leftrightarrow \text{false} \\ \text{(S) chp last:} \quad \mathbf{last}; \psi \leftrightarrow \psi \\ \text{(S) chp last 2:} \quad \varphi; \mathbf{last} \leftrightarrow \varphi \\ \text{(S) chp ass:} \quad (\varphi_1; \varphi_2); \psi \leftrightarrow \varphi_1; \varphi_2; \psi \\ \text{(S) lst chp:} \quad \mathbf{last} \rightarrow (\varphi; \psi \leftrightarrow \varphi \wedge \psi) \end{array}$

B.5.2 φ^* (star)

Syntax	$\varphi^* \in \mathbf{F}$, if $\varphi \in \mathbf{F}$
Semantics	

$I \models \varphi^*$ iff <ul style="list-style-type: none"> $I = 0$ or there exists $0 = n_0 < n_1 < \dots < n_m < I$ <ul style="list-style-type: none"> with $I _{n_i}^{n_{i+1}} \models \varphi$ for all $0 \leq i < m$ and $I _{n_m} \models \varphi$ or $I = \infty$ <ul style="list-style-type: none"> and there exists infinitely many $0 = n_0 < n_1 < \dots$ with $I _{n_i}^{n_{i+1}} \models \varphi$ for all $0 \leq i$
Axioms <i>star</i> : $\varphi^* \leftrightarrow \mathbf{last} \vee (\varphi \wedge \circ \mathbf{true}); \varphi^*$
Properties (S) <i>star last</i> : $\mathbf{last}^* \leftrightarrow \mathbf{last}$

B.5.3 step (atomic step)

Syntax	$\mathbf{step} \in \mathbf{F}$
Semantics	$I \models \mathbf{step}$ iff $ I = 1$

B.6 LTL Operators

B.6.1 φ until ψ (until)

Syntax	$\varphi \mathbf{until} \psi \in \mathbf{F}$, if $\varphi, \psi \in \mathbf{F}$
Semantics	$I \models \varphi \mathbf{until} \psi$ iff <ul style="list-style-type: none"> there exists $n \leq I$ with $I _n \models \psi$ and $I _m \models \varphi$ for all $0 \leq m < n$
Axioms	(I) <i>until</i> : $\varphi \mathbf{until} \psi \leftrightarrow \psi \vee \varphi \wedge \circ (\varphi \mathbf{until} \psi)$ <i>until live</i> : $\varphi \mathbf{until} \psi \rightarrow \exists l. \diamond l \wedge (\neg l \wedge \varphi) \mathbf{until} \psi$
Properties	<i>until lem 1</i> : $\square (\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1 \mathbf{until} \psi \rightarrow \varphi_2 \mathbf{until} \psi)$ <i>until lem 2</i> : $\square (\varphi_1 \rightarrow \varphi_2) \rightarrow (\psi \mathbf{until} \varphi_1 \rightarrow \psi \mathbf{until} \varphi_2)$ (S) <i>until true 1</i> : $\mathbf{true} \mathbf{until} \varphi \leftrightarrow \diamond \varphi$ (S) <i>until false 1</i> : $\mathbf{false} \mathbf{until} \varphi \leftrightarrow \varphi$

(S)	<i>until true 2:</i>	φ until true	\leftrightarrow	true
(S)	<i>until false 2:</i>	φ until false	\leftrightarrow	false
(I)	<i>until pos 1:</i>	$\varphi \rightarrow (\varphi$ until ψ	\leftrightarrow	$\psi \vee \circ (\varphi$ until $\psi))$
(S)	<i>until neg 1:</i>	$\neg \varphi \rightarrow (\varphi$ until ψ	\leftrightarrow	$\psi)$
(S)	<i>until pos 2:</i>	$\psi \rightarrow (\varphi$ until ψ	\leftrightarrow	true)
(I)	<i>until neg 2:</i>	$\neg \psi \rightarrow (\varphi$ until ψ	\leftrightarrow	$\varphi \wedge \circ (\varphi$ until $\psi))$
(S)	<i>lst until:</i>	last $\rightarrow (\varphi$ until ψ	\leftrightarrow	$\psi)$

B.6.2 $\diamond \varphi$ (eventually)

Syntax	$\diamond \varphi \in \mathbf{F}$, if $\varphi \in \mathbf{F}$
Semantics	$\diamond \varphi := \text{true until } \varphi$
Properties	
(I)	<i>ev:</i> $\diamond \varphi \leftrightarrow \varphi \vee \circ \diamond \varphi$
	<i>ev live:</i> $\diamond \varphi \leftrightarrow \exists l. \diamond l \wedge (\neg l) \text{ until } \varphi$
(S)	<i>ev pos:</i> $\varphi \rightarrow (\diamond \varphi \leftrightarrow \text{true})$

B.6.3 $\square \varphi$ (always)

Syntax	$\square \varphi \in \mathbf{F}$, if $\varphi \in \mathbf{F}$
Semantics	$\square \varphi := \neg \diamond \neg \varphi$
Properties	
(I)	<i>alw:</i> $\square \varphi \leftrightarrow \varphi \wedge \bullet \square \varphi$
	<i>alw safe:</i> $\square \varphi \leftrightarrow \forall l. \diamond l \rightarrow \varphi \text{ unless } l$
	<i>alw pos:</i> $\varphi \rightarrow (\square \varphi \leftrightarrow \bullet \square \varphi)$

B.6.4 φ unless ψ (unless)

Syntax	φ unless $\psi \in \mathbf{F}$, if $\varphi, \psi \in \mathbf{F}$
Semantics	φ unless $\psi := \varphi$ until $\psi \vee \square \varphi$
Properties	
	<i>unls lem 1:</i> $\square (\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1 \text{ unless } \psi \rightarrow \varphi_2 \text{ unless } \psi)$

	<i>unls lem 2:</i>	$\Box (\varphi_1 \rightarrow \varphi_2) \rightarrow (\psi \text{ unless } \varphi_1 \rightarrow \psi \text{ unless } \varphi_2)$
(I)	<i>unls:</i>	$\varphi \text{ unless } \psi \leftrightarrow \psi \vee \varphi \wedge \bullet (\varphi \text{ unless } \psi)$
	<i>unls safe:</i>	$\varphi \text{ unless } \psi \leftrightarrow \forall l. \Diamond l \rightarrow \varphi \text{ unless } (\psi \vee l)$
(S)	<i>unls true 1:</i>	$\text{true unless } \varphi \leftrightarrow \text{true}$
(S)	<i>unls false 1:</i>	$\text{false unless } \varphi \leftrightarrow \varphi$
(S)	<i>unls true 2:</i>	$\varphi \text{ unless true} \leftrightarrow \text{true}$
(S)	<i>unls false 2:</i>	$\varphi \text{ unless false} \leftrightarrow \Box \varphi$
(I)	<i>unls pos 1:</i>	$\varphi \rightarrow (\varphi \text{ unless } \psi \leftrightarrow \psi \vee \bullet (\varphi \text{ unless } \psi))$
(S)	<i>unls neg 1:</i>	$\neg \varphi \rightarrow (\varphi \text{ unless } \psi \leftrightarrow \psi)$
(S)	<i>unls pos 2:</i>	$\psi \rightarrow (\varphi \text{ unless } \psi \leftrightarrow \text{true})$
(I)	<i>unls neg 2:</i>	$\neg \psi \rightarrow (\varphi \text{ until } \psi \leftrightarrow \varphi \wedge \bullet (\varphi \text{ unless } \psi))$
(S)	<i>lst unls:</i>	$\text{last} \rightarrow (\varphi \text{ unless } \psi \leftrightarrow \varphi \vee \psi)$

B.7 Program Quantifiers

B.7.1 $\exists x. \varphi$ (hiding)

Syntax	$\exists x. \varphi \in \mathbf{F}, \text{ if } x \in \mathbf{Z} \setminus \{\text{blk}\}, \varphi \in \mathbf{F}$
Semantics	$I \models \exists x. \varphi \text{ iff there exists } I_0 \text{ with } I_0 =_x I \text{ and } I_0 \models \varphi$
Axioms	<p><i>exx rw:</i> $(\forall x. \varphi_1 \leftrightarrow \varphi_2) \rightarrow (\exists x. \varphi_1 \leftrightarrow \exists x. \varphi_2)$</p> <p><i>exx lem:</i> $(\forall x. \varphi_1 \rightarrow \varphi_2) \rightarrow (\exists x. \varphi_1 \rightarrow \exists x. \varphi_2)$</p> <p><i>exx dis:</i> $(\exists x. \varphi_1 \vee \varphi_2) \leftrightarrow (\exists x. \varphi_1) \vee (\exists x. \varphi_2)$</p> <p><i>exx ex:</i> $(\exists x. \exists v. \varphi) \leftrightarrow \exists v. \exists x. \varphi$</p> <p>(S) <i>exx lst:</i> $(\exists x. \rho \wedge \text{last}) \leftrightarrow (\exists X_0. \rho^{[X_0, X_0, X_0/x, x', x'']}) \wedge \text{last}$ X_0 fresh with respect to $\text{free}(\rho)$</p> <p>(S) <i>exx stp:</i> $(\exists x. \rho \wedge \delta \wedge \beta \wedge \circ \varphi) \leftrightarrow \exists X_2. (\exists X_0, X_1. \rho_0) \wedge \delta_0 \wedge \beta \wedge \circ \exists x. x = X_2 \wedge \varphi$ $\rho_0 := \text{frm}(\rho, \delta)^{[X_0, X_1, X_2/x, x', x'']}$</p>

$\delta_0 := \begin{cases} [\vec{x} \cup \{x\}] & \delta \equiv [\vec{x}] \\ X_1 \neq X_0 \vee \neg [\vec{x} \cup \{x\}] & \delta \equiv \neg [\vec{x}], x \notin \vec{x} \\ \delta & \text{otherwise} \end{cases}$ <p>X_0, X_1, X_2 fresh with respect to $\text{free}(\rho, \varphi)$</p>
<p>Properties</p> <p style="text-align: center;"><i>exx elim:</i> $(\exists x. \varphi) \leftrightarrow \varphi$ where $x \notin \text{free}(\varphi)$</p>

B.7.2 $\forall x. \varphi$ (universal hiding)

Syntax	$\forall x. \varphi \in \mathbf{F}$, if $x \in \mathbf{Z} \setminus \{\text{blk}\}, \varphi \in \mathbf{F}$
Semantics	$\forall x. \varphi := \neg \exists x. \neg \varphi$
<p>Properties</p> <p><i>all rw:</i> $(\forall x. \varphi_1 \leftrightarrow \varphi_2) \rightarrow (\forall x. \varphi_1 \leftrightarrow \forall x. \varphi_2)$</p> <p><i>all lem:</i> $(\forall x. \varphi_1 \rightarrow \varphi_2) \rightarrow (\forall x. \varphi_1 \rightarrow \forall x. \varphi_2)$</p> <p><i>all con:</i> $(\forall x. \varphi_1 \wedge \varphi_2) \leftrightarrow (\forall x. \varphi_1) \wedge (\forall x. \varphi_2)$</p> <p><i>all all:</i> $(\forall x. \forall v. \varphi) \leftrightarrow \forall v. \forall x. \varphi$</p> <p>(S) <i>all lst:</i> $(\forall x. \rho \vee \neg \mathbf{last}) \leftrightarrow (\forall X_0. \rho^{[X_0, X_0, X_0/x, x', x'']}) \vee \neg \mathbf{last}$ X_0 fresh with respect to $\text{free}(\rho)$</p> <p>(S) <i>all stp:</i> $(\forall x. (\rho \vee \delta \vee \beta \vee \bullet \varphi)) \leftrightarrow \forall X_2. (\forall X_0, X_1. \rho_0) \vee \delta_0 \vee \beta \vee \bullet \forall x. x = X_2 \rightarrow \varphi$ $\rho_0 := \text{frm}(\rho, \delta)^{[X_0, X_1, X_2/x, x', x'']}$ $\delta_0 := \begin{cases} \neg [\vec{x} \cup \{x\}] & \delta \equiv \neg [\vec{x}] \\ X_1 = X_0 \wedge [\vec{x} \cup \{x\}] & \delta \equiv [\vec{x}], x \notin \vec{x} \\ \delta & \text{otherwise} \end{cases}$ X_0, X_1, X_2 fresh with respect to $\text{free}(\rho, \varphi)$</p>	

B.8 Sequential Programs

B.8.1 $x := e$ (assignment)

Syntax	$x := e \in \mathbf{F}$, if $x \in \mathbf{Z}_s \setminus \{\text{blk}\}, e \in \mathbf{E}_s$
Semantics	$x := e := x' = e \wedge [x] \wedge \circ \mathbf{last}$
Properties	

<i>asg rw</i> : $e_1 = e_2 \rightarrow (\quad x := e_1$ $\quad \leftrightarrow x := e_2)$
<i>asg</i> : $x := e \leftrightarrow x' = e \wedge [x] \wedge \neg \mathbf{blocked} \wedge \circ \mathbf{last}$

B.8.2 $x := ?$ (random assignment)

Syntax $x := ? \in \mathbf{F}$, if $x \in \mathbf{Z}$
Semantics $x := ? \equiv [x] \wedge \circ \mathbf{last}$
Properties <i>rasg</i> : $x := ? \leftrightarrow \mathbf{true} \wedge [x] \wedge \neg \mathbf{blocked} \wedge \circ \mathbf{last}$

B.8.3 skip (no operation)

Syntax $\mathbf{skip} \in \mathbf{F}$
Semantics $\mathbf{skip} \equiv [] \wedge \circ \mathbf{last}$
Properties <i>skp</i> : $\mathbf{skip} \leftrightarrow \mathbf{true} \wedge [] \wedge \neg \mathbf{blocked} \wedge \circ \mathbf{last}$

B.8.4 $\mathbf{var} \ x = e \ \mathbf{in} \ \varphi$ (local variable declaration)

Syntax $\mathbf{var} \ x = e \ \mathbf{in} \ \varphi$, if $x \in \mathbf{Z}_s \setminus \{\mathbf{blk}\}, e \in \mathbf{E}_s$
Semantics $\mathbf{var} \ x = e \ \mathbf{in} \ \varphi \equiv \quad \square x' = x$ $\quad \wedge \exists X. X = e \wedge \exists x. x = X \wedge \varphi \wedge \square x'' = x'$ $X \notin \mathbf{free}(e, \varphi)$
Properties <i>var lem</i> : $(\forall x. \varphi_1 \rightarrow \varphi_2) \rightarrow (\quad \mathbf{var} \ x = e \ \mathbf{in} \ \varphi_1$ $\quad \rightarrow \mathbf{var} \ x = e \ \mathbf{in} \ \varphi_2)$ <i>var dis</i> : $\mathbf{var} \ x = e \ \mathbf{in} \ (\varphi_1 \vee \varphi_2)$ $\leftrightarrow \mathbf{var} \ x = e \ \mathbf{in} \ \varphi_1 \vee \mathbf{var} \ x = e \ \mathbf{in} \ \varphi_2$ <i>var ex</i> : $\mathbf{var} \ x = e \ \mathbf{in} \ (\exists v. \varphi) \leftrightarrow \exists v_0. \mathbf{var} \ x = e \ \mathbf{in} \ \varphi^{[v_0/v]}$ v_0 fresh with respect to $(\mathbf{free}(\varphi) \setminus \{v\}) \cup \mathbf{free}(e)$ (S) <i>var lst</i> : $\mathbf{var} \ x = e \ \mathbf{in} \ (\rho \wedge \mathbf{last}) \leftrightarrow \rho^{[e, e / x, x', x']}$ (S) <i>var stp</i> : $\mathbf{var} \ x = e \ \mathbf{in} \ (\rho \wedge \delta \wedge \beta \wedge \circ \varphi)$ $\leftrightarrow \exists X_0. (\rho_0 \wedge \delta_0 \wedge \beta \wedge \circ (\mathbf{var} \ x = e_0 \ \mathbf{in} \ \varphi))$ (S) <i>var wstp</i> : $\mathbf{var} \ x = e \ \mathbf{in} \ (\rho \wedge \delta \wedge \beta \wedge \bullet \varphi)$ $\leftrightarrow \exists X_0. (\rho_0 \wedge \delta_0 \wedge \beta \wedge \bullet (\mathbf{var} \ x = e_0 \ \mathbf{in} \ \varphi))$ X_0 fresh with respect to $\mathbf{free}(e, \rho, \varphi)$

$\rho_0 := \text{frm}(\rho[x'/x''], \delta)[e, X_0/x, x']$ $\delta_0 := \begin{cases} [\vec{x} \setminus \{x\}], & \text{if } \delta \equiv [\vec{x}] \\ x' = x \wedge \delta, & \text{otherwise} \end{cases}$ $e_0 := \text{frm}(x', \delta)[e, X_0/x, x']$ <p><i>var xtrct:</i> var x = e in φ $\leftrightarrow \exists x_0. x_0 = e \wedge \mathbf{var\ x = x_0\ in\ } (\Box x' = x'_0 \wedge \varphi)$ $x_0 \in \mathbf{Y}$ fresh with respect to $\text{free}(e, \varphi)$</p>
--

B.8.5 var x = ? in φ (random variable declaration)

Syntax	var x = ? in φ , if $x \in \mathbf{Z} \setminus \{\text{blk}\}$
Semantics	var x = ? in $\varphi := \Box x' = x \wedge \exists x. \varphi \wedge \Box x'' = x'$
Properties	
<i>rnd lem:</i>	$(\forall x. \varphi_1 \rightarrow \varphi_2) \rightarrow (\mathbf{var\ x = ?\ in\ } \varphi_1 \rightarrow \mathbf{var\ x = ?\ in\ } \varphi_2)$
<i>rnd dis:</i>	$\mathbf{var\ x = ?\ in\ } (\varphi_1 \vee \varphi_2) \leftrightarrow \mathbf{var\ x = ?\ in\ } \varphi_1 \vee \mathbf{var\ x = ?\ in\ } \varphi_2$
<i>rnd ex:</i>	$\mathbf{var\ x = ?\ in\ } (\exists v. \varphi) \leftrightarrow \exists v. \mathbf{var\ x = ?\ in\ } \varphi$
(S) <i>rnd lst:</i>	$\mathbf{var\ x = ?\ in\ } (\rho \wedge \text{last}) \leftrightarrow \exists X_0. \rho[X_0, X_0, X_0/x, x', x'']$ X_0 fresh with respect to $\text{free}(\rho)$
(S) <i>rnd stp:</i>	$\mathbf{var\ x = ?\ in\ } (\rho \wedge \delta \wedge \beta \wedge \circ \varphi) \leftrightarrow \exists X_0, X_1. (\rho_0 \wedge \delta_0 \wedge \beta \wedge \circ (\mathbf{var\ x = e_0\ in\ } \varphi))$
(S) <i>rnd wstp:</i>	$\mathbf{var\ x = ?\ in\ } (\rho \wedge \delta \wedge \beta \wedge \bullet \varphi) \leftrightarrow \exists X_0, X_1. (\rho_0 \wedge \delta_0 \wedge \beta \wedge \bullet (\mathbf{var\ x = e_0\ in\ } \varphi))$ X_0, X_1 fresh with respect to $\text{free}(e, \rho, \varphi)$ $\rho_0 := \text{frm}(\rho[x'/x''], \delta)[X_0, X_1/x, x']$ $\delta_0 := \begin{cases} [\vec{x} \setminus \{x\}], & \text{if } \delta \equiv [\vec{x}] \\ x' = x \wedge \delta, & \text{otherwise} \end{cases}$ $e_0 := \text{frm}(x', \delta)[X_0, X_1/x, x']$

B.8.6 if ψ then φ_1 else φ_2 (conditional)

Syntax	if ψ then φ_1 else $\varphi_2 \in \mathbf{F}$, if $\psi, \varphi_1, \varphi_2 \in \mathbf{F}$
Semantics	if ψ then φ_1 else $\varphi_2 := (\psi \rightarrow \varphi_1) \wedge (\neg \psi \rightarrow \varphi_2)$
Properties	

(E) <i>ite</i> :	if ψ then φ_1 else φ_2	\leftrightarrow	$\psi \wedge \varphi_1$ $\vee \neg \psi \wedge \varphi_2$
(E) <i>ite cnf</i> :	if ψ then φ_1 else φ_2	\leftrightarrow	$(\psi \rightarrow \varphi_1)$ $\wedge (\neg \psi \rightarrow \varphi_2)$
(S) <i>ite false 1</i> :	if ψ then false else φ_2	\leftrightarrow	$\neg \psi \wedge \varphi_2$
(S) <i>ite false 2</i> :	if ψ then φ_1 else false	\leftrightarrow	$\neg \psi \wedge \varphi_1$
(S) <i>ite pos</i> :	$\psi \rightarrow$ (if ψ then φ_1 else φ_2)	\leftrightarrow	φ_1
(S) <i>ite neg</i> :	$\neg \psi \rightarrow$ (if ψ then φ_1 else φ_2)	\leftrightarrow	φ_2

B.8.7 if ψ then φ (simple conditional)

Syntax	if ψ then $\varphi \in \mathbf{F}$, if $\psi, \varphi \in \mathbf{F}$
Semantics	if ψ then φ $:\equiv$ if ψ then φ else last

B.8.8 while ψ do φ (loop)

Syntax	while ψ do $\varphi \in \mathbf{F}$, if $\varphi, \psi \in \mathbf{F}$
Semantics	while ψ do φ $:\equiv$ $(\psi \wedge \varphi)^* \wedge$ finally $\neg \psi$
Properties	
	<i>whl lem 1</i> : $[\text{true}] \bullet (\varphi_1 \rightarrow \varphi_2) \rightarrow$ (while φ_1 do φ \rightarrow while φ_2 do ψ)
	<i>whl lem 2</i> : $[\text{true}] \bullet (\varphi_1 \rightarrow \varphi_2) \rightarrow$ (while ψ do φ_1 \rightarrow while ψ do φ_2)
(I) <i>whl</i> :	while ψ do φ \leftrightarrow $(\psi \wedge \varphi \wedge \circ \text{true});$ while ψ do φ $\vee \neg \psi \wedge$ last
(I) <i>whl cnf</i> :	while ε do φ \leftrightarrow $(\varepsilon \rightarrow (\varphi \wedge \circ \text{true});$ while ε do φ) $\wedge (\neg \varepsilon \rightarrow$ last)
(I) <i>whl pos</i> :	$\varepsilon \rightarrow$ (while ε do φ \leftrightarrow $(\varphi \wedge \circ \text{true});$ while ε do φ)
(S) <i>whl neg</i> :	$\neg \varepsilon \rightarrow$ (while ε do φ \leftrightarrow last)
(S) <i>lst whl</i> :	last \rightarrow (while ψ do φ \leftrightarrow $\neg \psi$)

B.8.9 abort (nonterminating program)

Syntax	abort $\in \mathbf{F}$
Semantics	abort $:\equiv$ while true do skip
Properties	
(I) <i>abrt</i> :	abort \leftrightarrow true \wedge \square \wedge blocked \wedge \circ abort
(S) <i>lst abrt</i> :	last \rightarrow (abort \leftrightarrow false)

B.9 Synchronisation

B.9.1 await φ (synchronisation)

Syntax	await $\varphi \in \mathbf{F}$, if $\varphi \in \mathbf{F}$
Semantics	await φ $:\equiv$ while $\neg \varphi$ do (blocked \wedge \circ last)
Properties	
(I) <i>awt</i> :	await ε \leftrightarrow $\varepsilon \wedge$ last $\vee \neg \varepsilon \wedge \square \wedge$ blocked \wedge \circ await ε
(S) <i>awt pos</i> :	$\varepsilon \rightarrow$ (await ε \leftrightarrow last)
(I) <i>awt neg</i> :	$\neg \varepsilon \rightarrow$ (await ε \leftrightarrow true \wedge \square \wedge blocked \wedge \circ await ε)

B.10 Interleaving

B.10.1 $l_1 :: \varphi \parallel^< l_2 :: \psi$ (left interleaving)

Syntax	$l_1 :: \varphi \parallel^< l_2 :: \psi \in \mathbf{F}$, if $l_1, \varphi, l_2, \psi \in \mathbf{F}$
Semantics	
$I \models l_1 :: \varphi \parallel^< l_2 :: \psi$ iff there exist $I_1, I_2, n \in \mathbb{N}_0$ with $I \in \llbracket l_1 :: I_1 < \parallel_{n+1} l_2 :: I_2 \rrbracket$ and $I_1 \models \varphi$ and $I_2 \models \psi$	
$\frac{I_1 \xrightarrow{\sigma, \sigma'_1} \emptyset \quad I_2 \xrightarrow{\sigma, \sigma'_2} t'}{(l_1 :: I_1 < \parallel_{n+1} l_2 :: I_2) \xrightarrow{\sigma, \sigma'_2} t'} \text{ ilvl lst}$	

$\frac{I_1 \xrightarrow{\sigma_1, \sigma'_1} I'_1 \quad I_1 \text{ progresses} \quad (\sigma_1) \not\vdash l_2}{(l_1 :: I_1 < _{n+1} l_2 :: I_2) \xrightarrow{\sigma_1, \sigma'_1} (l_1 :: I'_1 < _n l_2 :: I_2)} \text{ ilvl stp}$ $\frac{I_1 \xrightarrow{\sigma, \sigma'_1} I'_1 \quad I_1 \text{ is blocked} \quad I_2 \xrightarrow{\sigma, \sigma'_2} I'_2}{(l_1 :: I_1 < _{n+1} l_2 :: I_2) \xrightarrow{\sigma, \sigma'_2} (l_1 :: I'_1 < _n l_2 :: I'_2)} \text{ ilvl blk}$ $\frac{(l_2 :: I_2 < _{n+1} l_1 :: I_1) \xrightarrow{\sigma, \sigma'} t'}{(l_1 :: I_1 < _0 l_2 :: I_2) \xrightarrow{\sigma, \sigma'} t'} \text{ ilvl switch}_n, \text{ for } n \in \mathbb{N}_0$
<p>Axioms</p> <p><i>ilvl lem:</i> $[\text{true}] (\varphi_1 \rightarrow \varphi_2) \rightarrow (l_1 :: \varphi_1 \ \! < l_2 :: \psi \rightarrow l_1 :: \varphi_2 \ \! < l_2 :: \psi)$</p> <p><i>ilvl dis:</i> $l_1 :: (\varphi_1 \vee \varphi_2) \ \! < l_2 :: \psi \leftrightarrow l_1 :: \varphi_1 \ \! < l_2 :: \psi \vee l_1 :: \varphi_2 \ \! < l_2 :: \psi$</p> <p><i>ilvl ex:</i> $l_1 :: (\exists v. \varphi) \ \! < l_2 :: \psi \leftrightarrow \exists v_0. l_1 :: \varphi^{[v_0/v]} \ \! < l_2 :: \psi$ v_0 fresh with respect to $(\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(l_1, l_2, \psi)$</p> <p>(S) <i>ilvl lst:</i> $l_1 :: (\rho \wedge \mathbf{last}) \ \! < l_2 :: \psi \leftrightarrow \rho^{[w,w/w',w'']}] \wedge \psi$</p> <p>(S) <i>ilvl stp:</i> $l_1 :: (\rho \wedge \delta \wedge \neg \mathbf{blocked} \wedge \circ \varphi) \ \! < l_2 :: \psi \leftrightarrow \exists X_2. (\rho^{[X_2/w'']}] \wedge \neg \text{trm}(l_2) \wedge \delta \wedge \neg \mathbf{blocked} \wedge \circ (l_1 :: (w = X_2 \wedge \varphi) \ \! l_2 :: \psi))$</p> <p>(S) <i>ilvl blk:</i> $l_1 :: (\rho \wedge \delta \wedge \mathbf{blocked} \wedge \circ \varphi) \ \! < l_2 :: \psi \leftrightarrow \exists X_2. (\exists X_1. \text{frm}(\rho, \delta)^{[X_1, X_2/w', w'']}] \wedge l_1 :: (w = X_2 \wedge \varphi) \ \! _b^< l_2 :: \psi$</p>
<p>Properties</p> <p>(S) <i>ilvl false 2:</i> $l_1 :: \text{false} \ \! < l_2 :: \psi \leftrightarrow \text{false}$</p> <p>(S) <i>ilvl false 4:</i> $l_1 :: \varphi \ \! < l_2 :: \text{false} \leftrightarrow \text{false}$</p> <p>(S) <i>ilvl true:</i> $\text{true} \ \! < \text{true} \leftrightarrow \text{true}$</p> <p>(S) <i>lst ilvl:</i> $\mathbf{last} \rightarrow (\varphi \ \! < \psi \leftrightarrow \varphi \wedge \psi)$</p>

B.10.2 $l_1 :: \varphi \|\!|_b^< l_2 :: \psi$ (blocked left interleaving)

Syntax	$l_1 :: \varphi \ \! _b^< l_2 :: \psi \in \mathbf{F}$, if $l_1, \varphi, l_2, \psi \in \mathbf{F}$
Semantics	$l_1 :: \varphi \ \! _b^< l_2 :: \psi \equiv l_1 :: (\mathbf{blocked} \wedge \circ \varphi) \ \! < l_2 :: \psi$
Axioms	

<i>ilvlb lem:</i>	[true] $(\varphi_1 \rightarrow \varphi_2) \rightarrow (l_2 :: \psi \parallel_b^< l_1 :: \varphi_1 \rightarrow l_2 :: \psi \parallel_b^< l_1 :: \varphi_2)$
<i>ilvlb dis:</i>	$l_2 :: \psi \parallel_b^< l_1 :: (\varphi_1 \vee \varphi_2)$ $\leftrightarrow l_2 :: \psi \parallel_b^< l_1 :: \varphi_1 \vee l_2 :: \psi \parallel_b^< l_1 :: \varphi_2$
<i>ilvlb ex:</i>	$l_2 :: \psi \parallel_b^< l_1 :: (\exists v. \varphi)$ $\leftrightarrow \exists v_0. l_2 :: \psi \parallel_b^< l_1 :: \varphi^{[v_0/v]}$ v_0 fresh with respect to $(\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(l_1, l_2, \psi)$
(S) <i>ilvlb lst:</i>	$l_2 :: \psi \parallel_b^< l_1 :: (\rho \wedge \mathbf{last}) \leftrightarrow \mathbf{false}$
(S) <i>ilvlb stp:</i>	$l_2 :: \psi \parallel_b^< l_1 :: (\rho \wedge \delta \wedge \beta \wedge \circ \varphi)$ $\leftrightarrow \exists X_2. (\rho^{[X_2/w'']} \wedge \delta \wedge \beta \wedge \circ (l_2 :: \psi \parallel_b^< l_1 :: (w = X_2 \wedge \varphi)))$

B.10.3 $l_1 :: \varphi \parallel^> l_2 :: \psi$ (right interleaving)

Syntax	$l_1 :: \varphi \parallel^> l_2 :: \psi \in \mathbf{F}$, if $l_1, \varphi, l_2, \psi \in \mathbf{F}$
Semantics	$l_1 :: \varphi \parallel^> l_2 :: \psi \equiv l_2 :: \psi \parallel^< l_1 :: \varphi$
Properties	<p><i>ilvr lem:</i> [true] $(\varphi_1 \rightarrow \varphi_2) \rightarrow (l_2 :: \psi \parallel^> l_1 :: \varphi_1 \rightarrow l_2 :: \psi \parallel^> l_1 :: \varphi_2)$</p> <p><i>ilvr dis:</i> $l_2 :: \psi \parallel^> l_1 :: (\varphi_1 \vee \varphi_2)$ $\leftrightarrow l_2 :: \psi \parallel^> l_1 :: \varphi_1 \vee l_2 :: \psi \parallel^> l_1 :: \varphi_2$</p> <p><i>ilvr ex:</i> $l_2 :: \psi \parallel^> l_1 :: (\exists v. \varphi)$ $\leftrightarrow \exists v_0. l_2 :: \psi \parallel^> l_1 :: \varphi^{[v_0/v]}$ v_0 fresh with respect to $(\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(l_1, l_2, \psi)$</p> <p>(S) <i>ilvr lst:</i> $l_2 :: \psi \parallel^> l_1 :: (\rho \wedge \mathbf{last}) \leftrightarrow \rho^{[w,w'/w'',w''']} \wedge \psi$</p> <p>(S) <i>ilvr stp:</i> $l_2 :: \psi \parallel^> l_1 :: (\rho \wedge \delta \wedge \neg \mathbf{blocked} \wedge \circ \varphi)$ $\leftrightarrow \exists X_2. (\rho^{[X_2/w''']} \wedge \neg \text{trm}(l_2)) \wedge \delta \wedge \neg \mathbf{blocked} \wedge \circ (l_2 :: \psi \parallel_b^> l_1 :: (w = X_2 \wedge \varphi)))$</p> <p>(S) <i>ilvr blk:</i> $l_2 :: \psi \parallel^> l_1 :: (\rho \wedge \delta \wedge \mathbf{blocked} \wedge \circ \varphi)$ $\leftrightarrow \exists X_2. (\exists X_1. \text{frm}(\rho, \delta)^{[X_1, X_2/w', w''']}) \wedge l_2 :: \psi \parallel_b^> l_1 :: (w = X_2 \wedge \varphi)$</p>

B.10.4 $l_1 :: \varphi \parallel_b^> l_2 :: \psi$ (blocked right interleaving)

Syntax	$l_1 :: \varphi \parallel_b^> l_2 :: \psi \in \mathbf{F}$, if $l_1, \varphi, l_2, \psi \in \mathbf{F}$
Semantics	$l_1 :: \varphi \parallel_b^> l_2 :: \psi \equiv l_1 :: \varphi \parallel^> l_2 :: (\mathbf{blocked} \wedge \circ \psi)$

Properties	
<i>ilvrb lem:</i>	$[\text{true}] (\varphi_1 \rightarrow \varphi_2) \rightarrow (l_1 :: \varphi_1 \parallel_b^> l_2 :: \psi \rightarrow l_1 :: \varphi_2 \parallel_b^> l_2 :: \psi)$
<i>ilvrb dis:</i>	$l_1 :: (\varphi_1 \vee \varphi_2) \parallel_b^> l_2 :: \psi \leftrightarrow l_1 :: \varphi_1 \parallel_b^> l_2 :: \psi \vee l_1 :: \varphi_2 \parallel_b^> l_2 :: \psi$
<i>ilvrb ex:</i>	$l_1 :: (\exists v. \varphi) \parallel_b^> l_2 :: \psi \leftrightarrow \exists v_0. l_1 :: \varphi[v_0/v] \parallel_b^> l_2 :: \psi$ v_0 fresh with respect to $(\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(l_1, l_2, \psi)$
(S) <i>ilvrb lst:</i>	$l_1 :: (\rho \wedge \mathbf{last}) \parallel_b^> l_2 :: \psi \leftrightarrow \text{false}$
(S) <i>ilvrb stp:</i>	$l_1 :: (\rho \wedge \delta \wedge \beta \wedge \circ \varphi) \parallel_b^> l_2 :: \psi \leftrightarrow \exists X_2. (\rho[X_2/w] \wedge \delta \wedge \beta \wedge \circ (l_1 :: (w = X_2 \wedge \varphi) \parallel l_2 :: \psi))$

B.10.5 $l_1 :: \varphi \parallel l_2 :: \psi$ (interleaving)

Syntax	$l_1 :: \varphi \parallel l_2 :: \psi \in \mathbf{F}, \text{ if } l_1, \varphi, l_2, \psi \in \mathbf{F}$
Semantics	$l_1 :: \varphi \parallel l_2 :: \psi \equiv l_1 :: \varphi \parallel^< l_2 :: \psi \vee l_1 :: \varphi \parallel^> l_2 :: \psi$
Properties	
<i>ilv lem 1:</i>	$[\text{true}] \bullet (\varphi_1 \rightarrow \varphi_2) \rightarrow (l_1 :: \varphi_1 \parallel l_2 :: \psi \rightarrow l_1 :: \varphi_2 \parallel l_2 :: \psi)$
<i>ilv lem 2:</i>	$[\text{true}] \bullet (\varphi_1 \rightarrow \varphi_2) \rightarrow (l_2 :: \psi \parallel l_1 :: \varphi_1 \rightarrow l_2 :: \psi \parallel l_1 :: \varphi_2)$
(E) <i>ilv:</i>	$l_1 :: \varphi \parallel l_2 :: \psi \leftrightarrow l_1 :: \varphi \parallel^< l_2 :: \psi \vee l_1 :: \varphi \parallel^> l_2 :: \psi$
<i>ilv live 1:</i>	$l_1 :: (\exists l. \diamond l \wedge \varphi) \parallel l_2 :: \psi \rightarrow \exists l_0. \diamond l_0 \wedge (l_0 \vee l_1) :: \varphi_0 \parallel l_2 :: \psi$
<i>ilv live 2:</i>	$l_2 :: \psi \parallel l_1 :: (\exists l. \diamond l \wedge \varphi) \rightarrow \exists l_0. \diamond l_0 \wedge l_2 :: \psi \parallel (l_0 \vee l_1) :: \varphi_0$ $l_0 \notin \text{free}(l_1, l_2, \psi) \cup (\text{free}(\varphi) \setminus \{l\})$ $\varphi_0 \equiv \varphi[l_0/l]$
<i>ilv fair 1:</i>	$l_1 :: \varphi \parallel l_2 :: \psi \leftrightarrow \exists l_0. \diamond l_0 \wedge (l_0 \vee l_1) :: \varphi \parallel l_2 :: \psi$
<i>ilv fair 2:</i>	$l_2 :: \psi \parallel l_1 :: \varphi \leftrightarrow \exists l_0. \diamond l_0 \wedge l_2 :: \psi \parallel (l_0 \vee l_1) :: \varphi$ $l_0 \notin \text{free}(l_1, \varphi, l_2, \psi)$

B.11 Nondeterministic choice

$\varphi \parallel \psi$ (nondeterministic choice)

Syntax	$\varphi \parallel \psi \in \mathbf{F}$, if $\varphi, \psi \in \mathbf{F}$
Semantics	
$I \models \varphi \parallel \psi \quad \text{iff} \quad \begin{array}{l} \text{there exists } I_1, I_2 \\ \text{with } I \in \llbracket I_1 \parallel I_2 \rrbracket \text{ and } I_1 \models \varphi \text{ and } I_2 \models \psi \end{array}$	
$\frac{I_1 \xrightarrow{\sigma, \sigma'} t' \quad I_1 \text{ is active}}{I_1 \parallel I_2 \xrightarrow{\sigma, \sigma'} t'} \quad \text{chs 1} \quad \frac{I_2 \xrightarrow{\sigma, \sigma'} t' \quad I_2 \text{ is active}}{I_1 \parallel I_2 \xrightarrow{\sigma, \sigma'} t'} \quad \text{chs 2}$	
$\frac{I_1 \xrightarrow{\sigma, \sigma'} I'_1 \quad I_2 \xrightarrow{\sigma, \sigma'} I'_2 \quad I_1 \text{ is blocked} \quad I_2 \text{ is blocked}}{I_1 \parallel I_2 \xrightarrow{\sigma, \sigma'} I'_1 \parallel I'_2} \quad \text{chs blk}$	
Axioms	
$(E) \quad \text{chs: } \varphi \parallel \psi \leftrightarrow \neg \mathbf{blocked} \wedge \varphi \vee \neg \mathbf{blocked} \wedge \psi \vee \varphi \parallel^b \psi$	

$\varphi \parallel^b \psi$ (blocked nondeterministic choice)

Syntax	$\varphi \parallel^b \psi \in \mathbf{F}$, if $\varphi, \psi \in \mathbf{F}$
Semantics	
$\varphi \parallel^b \psi ::= (\mathbf{blocked} \wedge \varphi) \parallel (\mathbf{blocked} \wedge \psi)$	
Axioms	
$\text{chsb lem 1: } [\text{true}] (\mathbf{blocked} \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow (\varphi_1 \parallel^b \psi \rightarrow \varphi_2 \parallel^b \psi)$	
$\text{chsb lem 2: } [\text{true}] (\mathbf{blocked} \rightarrow (\varphi_1 \rightarrow \varphi_2)) \rightarrow (\psi \parallel^b \varphi_1 \rightarrow \psi \parallel^b \varphi_2)$	
$\text{chsb dis 1: } (\varphi_1 \vee \varphi_2) \parallel^b \psi \leftrightarrow \varphi_1 \parallel^b \psi \vee \varphi_2 \parallel^b \psi$	
$\text{chsb dis 2: } \psi \parallel^b (\varphi_1 \vee \varphi_2) \leftrightarrow \psi \parallel^b \varphi_1 \vee \psi \parallel^b \varphi_2$	
$\text{chsb ex 1: } (\exists v. \varphi) \parallel^b \psi \leftrightarrow \exists v_0. \varphi^{[v_0/v]} \parallel^b \psi$	
$\text{chsb ex 2: } \psi \parallel^b (\exists v. \varphi) \leftrightarrow \exists v_0. \psi \parallel^b \varphi^{[v_0/v]}$	
$v_0 \text{ fresh with respect to } (\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(\psi)$	
$(S) \quad \text{chsb ss: } \begin{array}{l} (\rho_1 \wedge \delta_1 \wedge \mathbf{blocked} \wedge \circ \varphi_1) \\ \parallel (\rho_2 \wedge \delta_2 \wedge \mathbf{blocked} \wedge \circ \varphi_2) \\ \leftrightarrow ((\rho_1 \wedge \rho_2) \wedge (\delta_1 \wedge \delta_2) \wedge \mathbf{blocked} \wedge \circ (\varphi_1 \parallel \varphi_2)) \end{array}$	

B.12 Atomic steps

$\{\varphi\}$ (atomic step)

Syntax	$\{\varphi\} \in \mathbf{F}$, if $\varphi \in \mathbf{F}$
Semantics	
$I \models \{\varphi\}$	iff $ I = 1$ and there exists I_0 with $ I_0 < \infty$ and $I_0(0) = I(0)$ and $I_0 \models \varphi$ and $I_0(I_0) = I(0)'$ and $I_0(m)' = I_0(m)''$ for all $m < I_0 $
Axioms	
<i>atm lem:</i>	$[\text{true}] (\varphi_1 \rightarrow \varphi_2) \rightarrow (\{\varphi_1\} \rightarrow \{\varphi_2\})$
<i>atm dis:</i>	$\{\varphi_1 \vee \varphi_2\} \leftrightarrow \{\varphi_1\} \vee \{\varphi_2\}$
<i>atm ex:</i>	$\{\exists v. \varphi\} \leftrightarrow \exists v. \{\varphi\}$
(S) <i>atm lst:</i>	$\{(\rho \wedge \mathbf{last})\} \leftrightarrow (\text{trm}(\rho) \wedge \square \wedge \neg \mathbf{blocked} \wedge \circ \mathbf{last})$
<i>atms lem:</i>	$[\text{true}] \bullet (\varphi_1 \rightarrow \varphi_2) \rightarrow (\{(\rho \wedge \delta \wedge \beta \wedge \circ \varphi_1)\} \rightarrow \{(\rho \wedge \delta \wedge \beta \wedge \circ \varphi_2)\})$
<i>atms dis:</i>	$\{(\rho \wedge \delta \wedge \beta \wedge \circ (\varphi_1 \vee \varphi_2))\} \leftrightarrow \{(\rho \wedge \delta \wedge \beta \wedge \circ \varphi_1)\} \vee \{(\rho \wedge \delta \wedge \beta \wedge \circ \varphi_2)\}$
<i>atms ex:</i>	$\{(\rho \wedge \delta \wedge \beta \wedge \circ (\exists v. \varphi))\} \leftrightarrow \exists v_0. \{(\rho \wedge \delta \wedge \beta \wedge \circ \varphi^{v_0/v})\}$ v_0 fresh with respect to $(\text{free}(\varphi) \setminus \{v\}) \cup \text{free}(\rho)$
(S) <i>atms lst:</i>	$\{(\rho_1 \wedge \delta \wedge \beta \wedge \circ (\rho_2 \wedge \mathbf{last}))\} \leftrightarrow (\rho_1[w'/w''] \wedge \rho_2[w',w'/w,w''] \wedge \delta \wedge \beta \wedge \circ \mathbf{last})$
(S) <i>atms stp:</i>	$\{(\rho_1 \wedge \delta_1 \wedge \beta_1 \wedge \circ (\rho_2 \wedge \delta_2 \wedge \beta_2 \wedge \circ \varphi))\} \leftrightarrow \{((\exists X. \rho_3 \wedge \rho_4) \wedge \delta_0 \wedge \beta_0 \wedge \circ \varphi)\}$ $\rho_3 \equiv \begin{cases} \text{frm}(\rho_1[w'/w''], \delta_1), & \text{if } \delta_2 \equiv [\vec{x}_2] \text{ and } w \notin \vec{x}_2 \\ \text{frm}(\rho_1[w'/w''], \delta_1)[X/w'], & \text{otherwise} \end{cases}$ $\rho_4 \equiv \begin{cases} \text{frm}^{-1}(\rho_2, \delta_2), & \text{if } \delta_1 \equiv [\vec{x}_1] \text{ and } w \notin \vec{x}_1 \\ \text{frm}^{-1}(\rho_2, \delta_2)[X/w], & \text{otherwise} \end{cases}$ $\delta_0 \equiv \begin{cases} [\vec{x}_1 \cup \vec{x}_2], & \text{if } \delta_1 \equiv [\vec{x}_1] \text{ and } \delta_2 \equiv [\vec{x}_2] \\ \text{true}, & \text{otherwise} \end{cases}$

$$\beta_0 := \begin{cases} \text{true} , & \text{if } \beta_1 \equiv \text{true} \text{ or } \beta_2 \equiv \text{true} \\ \neg \text{blocked} , & \text{if } \beta_1 \equiv \beta_2 \\ \text{blocked} , & \text{otherwise} \end{cases}$$

B.13 Labels

$l : \varphi$ (label)

Syntax	$l : \varphi \in \mathbf{F}$, if $l \in \mathbf{Z}_{\text{bool}} \setminus \{\text{blk}\}, \varphi \in \mathbf{F}$
Semantics	$l : \varphi := \blacksquare l' \neq l \wedge \exists l. \varphi$
Properties	
	<i>lbl lem:</i> $(\forall l. \varphi_1 \rightarrow \varphi_2) \rightarrow (l : \varphi_1 \rightarrow l : \varphi_2)$
	<i>lbl dis:</i> $l : (\varphi_1 \vee \varphi_2) \leftrightarrow l : \varphi_1 \vee l : \varphi_2$
	<i>lbl ex:</i> $l : (\exists v. \varphi) \leftrightarrow \exists v. l : \varphi$
(S)	<i>lbl lst:</i> $l : (\rho \wedge \mathbf{last}) \leftrightarrow ((\exists L_0. \rho^{[L_0, L_0, L_0 / l, l', l'']}) \wedge \mathbf{last})$ L_0 fresh with respect to $\text{free}(\rho)$
(S)	<i>lbl stp:</i> $l : (\rho \wedge \delta \wedge \beta \wedge \circ \varphi) \leftrightarrow \exists L_2. (l' \neq l \wedge \exists L_0, L_1. \rho_0 \wedge \delta_0 \wedge \beta \wedge \circ l : (l = L_2 \wedge \varphi))$ $\rho_0 := \text{frm}(\rho, \delta)^{[L_0, L_1, L_2 / l, l', l'']}$ $\delta_0 := \begin{cases} \lceil \vec{x} \cup \{1\} \rceil & \delta \equiv \lceil \vec{x} \rceil \\ L_1 \neq L_0 \vee \neg \lceil \vec{x} \cup \{1\} \rceil & \delta \equiv \neg \lceil \vec{x} \rceil, 1 \notin \vec{x} \\ \delta & \text{otherwise} \end{cases}$ L_0, L_1, L_2 fresh with respect to $\text{free}(\rho, \varphi)$

B.14 Procedures

$\text{proc}(e_1, \dots, e_n; x_1, \dots, x_m)$ (procedure)

Syntax	$\text{proc}(e_1, \dots, e_n; x_1, \dots, x_m) \in \mathbf{F}$, if $\text{proc} \in \mathbf{PROC}_{s_1^r, \dots, s_m^r}^r$, $e_i \in \mathbf{E}_{s_i}$, $x_j \in \mathbf{Z}_{s_j^r}$ with $x_i \neq x_j$
Semantics	

$\mathcal{A}, I \models \text{proc}(e_1, \dots, e_n; \mathbf{x}_1, \dots, \mathbf{x}_m)$ <p>iff</p> $\text{proc}_{\mathcal{A}} \left(\llbracket e_1 \rrbracket_{\mathcal{A}, I}, \dots, \llbracket e_n \rrbracket_{\mathcal{A}, I}, \left(d_{\mathbf{x}_1}^{(i)} \times \dots \times d_{\mathbf{x}_m}^{(i)} \times d_{\text{blk}}^{(i)} \right)_{i=0}^{2 \cdot I } \right)$ <p>and $\mathcal{A}, I \models \square \llbracket \mathbf{x}_1, \dots, \mathbf{x}_m, \text{blk} \rrbracket$</p>
<p>Axioms</p> <p><i>call lem val i:</i> $e_i^1 = e_i^2 \rightarrow (\text{proc}(e_1, \dots, e_i^1, \dots, e_n; \mathbf{x}_1, \dots, \mathbf{x}_m) \leftrightarrow \text{proc}(e_1, \dots, e_i^2, \dots, e_n; \mathbf{x}_1, \dots, \mathbf{x}_m))$</p> <p><i>call frm:</i> $\text{proc}(e_1, \dots, e_n; \mathbf{x}_1, \dots, \mathbf{x}_m) \rightarrow \llbracket \mathbf{x}_1, \dots, \mathbf{x}_m, \text{blk} \rrbracket$</p>

Appendix C

Proofs

C.1 Semantics

C.1.1 SOS relation for intervals (Corollary 2)

“ \Rightarrow ”: Case distinction over applicable SOS rules (see Def. 22).

Case “*trm*”: Assume $|I| = 0$, then $I(0)' = I(0)$ and $I|_1^\emptyset = \emptyset$.

Case “*stp*”: Assume $|I| > 0$, then $I|_1^\emptyset = I|_1$.

“ \Leftarrow ”

Case $|I| = 0$: Then $I(0)' = I(0)$ (see Def. 7) and $I|_1^\emptyset = \emptyset$ (see Def. 23). Therefore,

$$\frac{|I| = 0}{I \xrightarrow{I(0), I(0)'} I|_1^\emptyset} \text{trm}$$

Case $|I| > 0$: Then $I|_1^\emptyset = I|_1$. Therefore,

$$\frac{|I| > 0}{I \xrightarrow{I(0), I(0)'} I|_1^\emptyset} \text{stp}$$

□

C.1.2 Declarative semantics of interleaving

The semantics of left interleaving is defined as follows.

$$\begin{aligned}
& I \models l_1 :: \varphi \parallel^< l_2 :: \psi \\
\Leftrightarrow & \text{there exist } I_1, I_2, n \\
& \text{with } I \in \llbracket l_1 :: I_1 \parallel_{n+1} l_2 :: I_2 \rrbracket \\
& \text{and } I_1 \models \varphi \text{ and } I_2 \models \psi \\
\Leftrightarrow & \text{there exist } I_1, I_2, n, (t_i)_{i=0}^{|I|+1} \\
& \text{with } l_1 :: I_1 \parallel_{n+1} l_2 :: I_2 = t_0 \text{ and } (|I| < \infty \Rightarrow t_{|I|+1} = \emptyset) \\
& \text{and } t_i \xrightarrow{I^{(i)}, I^{(i)'}} t_{i+1} \text{ for all } i < \bar{n} \\
& \text{and } I_1 \models \varphi \text{ and } I_2 \models \psi
\end{aligned}$$

“ \Rightarrow ”: Take arbitrary but fixed $I_1, I_2, n, (t_i)_{i=0}^{|I|+1}$. Define a scheduling sequence $(p_i)_{i=0}^{|I|}$ with

$$\begin{aligned}
p_0 &= 1 \\
p_{i+1} &= \begin{cases} p_i, & \text{if there exists } l'_1, I'_1, l'_2, I'_2, n' \\ & \text{with } t_{i+1} = l'_1 :: I'_1 \parallel_{n'} l'_2 :: I'_2 \text{ with } n' \neq 0 \\ 3 - p_i, & \text{otherwise} \end{cases}
\end{aligned}$$

Counter n represents the number of steps before processes are switched. The SOS rules of Def. 24 ensure that n is decremented in every step until n is zero. The scheduling sequence $(p_i)_i$ starts out with scheduling process 1. Always, if counter n is zero, tasks are switched. After one of the two intervals I_1 or I_2 has terminated, the scheduling sequence alternates in every step. Thus, the scheduling sequence is guaranteed to be fair.

Using this scheduling sequence, the properties 1 - 6 of Lemma 1 can be directly derived from the premises of the SOS rules of Def. 24.

“ \Leftarrow ”: Let I_1, I_2 , and a fair scheduling sequence $(p_i)_{i=0}^{|I|}$ be given which satisfy the conditions of the lemma. Define a sequence of SOS terms $(t_i)_{i=0}^{|I|+1}$ as follows:

$$\begin{aligned}
\bar{n}_i &:= \min(\{\infty\} \cup \{j \mid p_{i+j} \neq p_i\}) - 1 \\
t_i &:= \begin{cases} l_1 :: I_1^{(i)} \parallel_{\bar{n}_i} l_2 :: I_2^{(i)}, & \text{if } p_i = 1 \\ l_2 :: I_2^{(i)} \parallel_{\bar{n}_i} l_1 :: I_1^{(i)}, & \text{otherwise} \end{cases}
\end{aligned}$$

where

$$\begin{aligned}
l_1 :: \emptyset \parallel_{\bar{n}} l_2 :: t_2 &= t_2 \\
l_1 :: t_1 \parallel_{\bar{n}} l_2 :: \emptyset &= t_1
\end{aligned}$$

Counter \bar{n}_i is set to the number of steps it takes for the scheduling sequence to switch tasks. As the scheduling sequence is fair, $\bar{n}_i = \infty$ only occurs after one of the two processes

has terminated. Depending on the current value of p_i , the SOS term either contains the first or the second interval on the left hand side. Special care has to be taken, if one of the intervals has terminated. It remains to show that the sequence of SOS terms satisfies the SOS relating of Def. 24. \square

For the following proofs, it is helpful to define a notion of program counters.

Definition 55 (*Program counters*) For scheduling two intervals I_1 and I_2 with a given scheduling sequence $(p_i)_{i=0}^{\bar{n}}$ two sequences of program counters $(pc_1^{(i)})_{i=0}^{\bar{n}+1}$ and $(pc_2^{(i)})_{i=0}^{\bar{n}+1}$ can be derived as follows.

$$pc_p^{(0)} = 0$$

$$pc_p^{(i+1)} = \begin{cases} pc_p^{(i)} + 1, & \text{if } p \in \pi_i \\ pc_p^{(i)}, & \text{otherwise} \end{cases}$$

The two sequences of program counters represent the number of transitions which have already been executed for the two interleaved processes. There is a direct connection between sequences of program counters and sequences of intervals as defined in Def. 26.

Lemma 35 (*Properties of program counters*) Given two intervals I_1 and I_2 and a scheduling sequence $(p_i)_{i=0}^{\bar{n}}$.

1. $I_1^{(i)} = I_1|_{pc_1^{(i)}}^{\emptyset}$
2. $I_2^{(i)} = I_2|_{pc_2^{(i)}}^{\emptyset}$

Directly follows from the definitions of scheduling intervals (Def. 26) and program counters (Def. 55). \square

Program counters link the states of intervals I_1 and I_2 to the states of the resulting interval I :

$$I(i) = I_1(pc_1^{(i)})$$

Inverse program counters as they are defined next, link the states of the resulting interval to the states of the intervals I_1 and I_2 :

$$I_1(i) = I((pc_1^{-1})^{(i)})$$

Definition 56 (*Inverse program counters*) Given two intervals I_1 and I_2 and a scheduling sequence $(p_i)_{i=0}^{\bar{n}}$. Two sequences $((pc_1^{-1})^{(i)})_{i=0}^{|I_1|}$ and $((pc_2^{-1})^{(i)})_{i=0}^{|I_2|}$ define the number of

the step where the i th step of the interval is scheduled.

$$\begin{aligned} (pc_p^{-1})^{(0)} &= \min \{j \mid p \in \pi_j\} \\ (pc_p^{-1})^{(i+1)} &= \min \left\{ j \mid (pc_p^{-1})^{(i)} < j \text{ and } p \in \pi_j \right\} \end{aligned}$$

Corollary 5 *Properties of inverse program counters*

$$1. \ pc_p^{((pc_p^{-1})^{(i)})} = i$$

C.1.3 Strong coincidence lemma (Theorem 1)

Assume two intervals I_1 and I_2 which satisfy the assumptions of the strong coincidence lemma. In order to show $\llbracket e \rrbracket_{I_1} = \llbracket e \rrbracket_{I_2}$, we use structural induction over expression e . Most of the cases are straightforward, and only the interesting cases are given here. Note that for formulas $e \in \mathbf{F}$ it is sufficient to show $I_1 \models e \Rightarrow I_2 \models e$ (compare Def. 8).

Case $e \equiv [x_1, \dots, x_n]$. We conclude

$$\begin{aligned} &I_1 \models [x_1, \dots, x_n] \\ \Leftrightarrow &\text{for all } x \in \mathbf{Z} \setminus \{x_1, \dots, x_n\}, \quad (\text{Def. 15}) \\ &I_1(0)'(x) = I_1(0)(x) \end{aligned}$$

With Def. 31, we receive $x \in \text{free}([x_1, \dots, x_n])$ for all $x \in \mathbf{Z} \setminus \{x_1, \dots, x_n\}$. Thus, with the second assumption of the strong coincidence theorem

$$I_1(0)'(x) = I_1(0)(x) \Leftrightarrow I_2(0)'(x) = I_2(0)(x)$$

and we continue

$$\begin{aligned} &\text{for all } x \in \mathbf{Z} \setminus \{x_1, \dots, x_n\}, \\ &I_1(0)'(x) = I_1(0)(x) \\ \Leftrightarrow &\text{for all } x \in \mathbf{Z} \setminus \{x_1, \dots, x_n\}, \\ &I_2(0)'(x) = I_2(0)(x) \\ \Leftrightarrow &I_2 \models [x_1, \dots, x_n] \quad (\text{Def. 15}) \end{aligned}$$

Case $e \equiv l_1 :: \varphi \parallel^< l_2 :: \psi$.

Assume $I_1 \models l_1 :: \varphi \parallel^< l_2 :: \psi$. According to the declarative semantics of interleaving, there are $I_{1,1}, I_{1,2}$, and a fair scheduling sequence $(p_i)_i$ which satisfy the conditions of Lemma 1. We define

$$\begin{aligned} I_{2,1}(i) &:= I_2((pc_1^{-1})^{(i)}) \\ I_{2,2}(i) &:= I_2((pc_2^{-1})^{(i)}) \end{aligned}$$

$$\begin{array}{rcccccccc}
I_1 & = & \sigma_{1,0} & \sigma'_{1,0} & \sigma''_{1,0} & \cdots & & \\
I_0 & = & \sigma_{1,0} & \sigma'_{0,0} & \sigma''_{0,0} & \sigma'_{0,1} & \sigma''_{0,1} & \cdots \\
& & \searrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
I_3 & = & \sigma_{2,0} & \sigma'_{3,0} & \sigma''_{3,0} & \sigma'_{3,1} & \sigma''_{3,1} & \cdots \\
& & \uparrow & \uparrow & & & & \\
I_2 & = & \sigma_{2,0} & \sigma'_{2,0} & \sigma''_{2,0} & \cdots & &
\end{array}$$

Figure C.1: Idea of how to construct I_3 in proof for strong coincidence lemma

It holds that $|I_1| = |I_2|$ and $I_1(\text{blk}) = I_2(\text{blk})$, because

$$\text{blk} \in \text{free}(l_1 :: \varphi \parallel^< l_2 :: \psi) .$$

Therefore, it is possible to leave the scheduling sequence unchanged. With $I_{2,1}$, $I_{2,2}$, and $(p_i)_i$, the conditions of Lemma 1 hold, because $I_{1,1}$ and $I_{2,1}$ (resp. $I_{1,2}$ and $I_{2,2}$) satisfy the requirements of the strong coincidence lemma and the induction hypothesis can be applied. Thus, we receive

$$I_2 \models l_1 :: \varphi \parallel^< l_2 :: \psi$$

Case $e \equiv \varphi \parallel \psi$.

A proof can be constructed similar to interleaving. However, a declarative semantics for nondeterministic choice must be defined in advance. This remains an open issue.

Case $e \equiv \langle \varphi \rangle \psi$. It follows that

$$\begin{aligned}
& I_1 \models \langle \varphi \rangle \psi \\
\Leftrightarrow & \text{there exists } I_0 && \text{(Def. 18)} \\
& \text{with } I_0(0) = I_1(0) \\
& \text{and } I_0 \models \varphi \text{ and } I_0 \models \psi
\end{aligned}$$

Given I_0 with $I_0(0) = I_1(0)$, we show

$$\begin{aligned}
& \text{there exists } I_3 && \text{(C.1)} \\
& \text{with } I_3(0) = I_2(0) \\
& \text{and } I_3 \models \varphi \text{ and } I_3 \models \psi
\end{aligned}$$

Without loss of generality $|I_1|, |I_2| > 0$ and $|I_0| > 1$. We construct I_3 with $I_3(0) = I_2(0)$ and $|I_3| = |I_0|$ as follows (see Fig. C.1): Interval I_3 is very similar to I_0 , except for the static variables and for the dynamic variables in the first two states. Static variables and the dynamic variables of the first state are equal to I_2 . The second state is a mixture of both second states of I_0 and I_2 with reference to the first state of I_0 . We define

$$\sigma'_{3,i}(x) := \sigma''_{3,i}(x) = \sigma_{2,0}(x) \quad \text{(C.2)}$$

$$\sigma'_{3,0}(y) := \begin{cases} \sigma'_{0,0}(y) & \text{if } y \in \text{vars}(\varphi) \cup \text{vars}(\psi) \\ \sigma_{2,0}(y) & \text{otherwise, if } \sigma'_{0,0}(y) = \sigma_{1,0}(y) \\ \sigma'_{0,0}(y) & \text{otherwise, if } \sigma_{2,0}(y) = \sigma_{1,0}(y) \\ \sigma_{1,0}(y) & \text{otherwise} \end{cases} \quad (\text{C.3})$$

$$\sigma'_{3,i+1}(y) := \sigma'_{0,i+1}(y) \quad (\text{C.4})$$

$$\sigma''_{3,i}(y) := \sigma''_{0,i}(y) \quad (\text{C.5})$$

To conclude $\llbracket \varphi \rrbracket_{I_0} = \llbracket \varphi \rrbracket_{I_3}$ and $\llbracket \psi \rrbracket_{I_0} = \llbracket \psi \rrbracket_{I_3}$ with the induction hypothesis, we must show:

1. $I_0(v) = I_3(v)$ for all $v \in \text{vars}(\varphi) \cup \text{vars}(\psi)$

If v is static, $I_0(v) = I_0(0)(v) = I_1(0)(v)$. Because $v \in \text{vars}(\varphi) \cup \text{vars}(\psi)$, also $v \in \text{vars}(\langle \varphi \rangle \psi)$, and we conclude with our initial assumptions for I_1, I_2 , that $I_1(0)(v) = I_2(0)(v) = I_3(v)$.

If v is dynamic, we show:

- $I_0(0)(v) = I_3(0)(v)$: in analogy to static case
- $I_0(0)'(v) = I_3(0)'(v)$: follows from (C.3) because of $v \in \text{vars}(\varphi) \cup \text{vars}(\psi)$.
- for all $0 < i < |I_0|$, $I_0(i)'(v) = I_3(i)'(v)$: follows from (C.4)
- for all $i < |I_0|$ $I_0(i)''(v) = I_3(i)''(v)$: follows from (C.5)

Therefore, $I_0(v) = I_3(v)$.

2. $I_0(n)'(y) = I_0(n)(y) \Leftrightarrow I_3(n)'(y) = I_3(n)(y)$ for all $y \in \text{free}(\varphi)$, $n < |I_0|$

Case $n = 0$: If $y \in \text{vars}(\varphi) \cup \text{vars}(\psi)$, then $I_0(y) = I_3(y)$ follows from Case 1 and property trivially holds. Assume $y \notin \text{vars}(\varphi) \cup \text{vars}(\psi)$. If $I_0(0)'(y) = I_0(0)(y)$, i.e., y stutters in first transition of I_0 , then $I_3(0)'(y) = I_3(0)(y) = I_2(0)(y)$ (C.3) and therefore I_3 also stutters. If $I_0(0)'(y) \neq I_0(0)(y)$, i.e., y does not stutter in first transition of I_0 , then the last two cases of (C.3) define $I_3(0)'(y)$ to be different from $I_3(0)(y)$.

Case $n > 0$: Because of (C.4) and (C.5), $I_0(n)'(y) = I_3(n)'(y)$ and $I_0(n)(y) = I_0(n-1)''(y) = I_3(n-1)''(y) = I_3(n)(y)$. Therefore, property trivially holds.

Thus, we have shown (C.1), which establishes $I_2 \models \langle \varphi \rangle \psi$. □

C.1.4 Substituting logical variables (Lemma 3)

An auxiliary definition on how to substitute intervals is helpful in the proof below.

Definition 57

$$\begin{aligned}
|I^{[\varepsilon/v]}| &= |I| \\
I^{[\varepsilon/v]}(v_0)(i) &= \begin{cases} \llbracket \varepsilon \rrbracket_{I(i)} , & \text{if } v \equiv v_0 \\ I(v_0)(i) , & \text{otherwise} \end{cases} \\
I^{[\varepsilon/v]}(v_0)(i)' &= \begin{cases} \llbracket \varepsilon \rrbracket_{I(i)'} , & \text{if } v \equiv v_0 \\ I(v_0)(i)' , & \text{otherwise} \end{cases} \\
I^{[\varepsilon/v]}(z) &= I(z)
\end{aligned}$$

Corollary 6 (*Properties of substituting intervals*)

$$1. I =_v I^{[\varepsilon/v]}$$

Directly follows from the Definition of $I^{[\varepsilon/v]}$ and $=_v$. □

Corollary 7 (*Properties of substituting logical variables in expressions*)

$$\llbracket e^{[\varepsilon/v]} \rrbracket_I = \llbracket e \rrbracket_{I^{[\varepsilon/v]}}$$

The proof is similar to the proof of the coincidence lemma in Appendix C.1.3. □

(Proof of Lemma 3.1)

$$\begin{aligned}
&\models \varphi^{[\varepsilon/v]} \\
\Leftrightarrow &\text{for all } I, I \models \varphi^{[\varepsilon/v]} \quad \text{Def. } \models \\
\Leftrightarrow &\text{for all } I, I^{[\varepsilon/v]} \models \varphi \quad \text{Cor. 7}
\end{aligned}$$

For an arbitrary I , show $I^{[\varepsilon/v]} \models \varphi$. This is fulfilled, because φ is valid. □

(Proof of Lemma 3.2)

$$\begin{aligned}
&I \models \varphi^{[\varepsilon/v]} \\
\Leftrightarrow &I^{[\varepsilon/v]} \models \varphi \quad \text{Cor. 7} \\
\Leftrightarrow &I \models \varphi \quad \text{Lem. 2}
\end{aligned}$$

The coincidence lemma can be applied, because $I =_v I^{[\varepsilon/v]}$ (see Cor. 6). □

(Proof of Lemma 3.3) Follows from the Semantics of \forall and Lemma 3.2. □

C.1.5 Renaming program variables (Lemma 4)

An auxiliary definition on how to rename program variables in intervals is helpful in the proof below.

Definition 58 (*Renaming intervals*) Let \vec{x} be a list of unique program variables. Let \vec{x}_0 be a permutation of \vec{x} . Define $I_{\vec{x}}^{\vec{x}_0}$ with

$$\begin{aligned} \left| I_{\vec{x}}^{\vec{x}_0} \right| &= |I| \\ I_{\vec{x}}^{\vec{x}_0}(X) &= I(X) \\ I_{\vec{x}}^{\vec{x}_0}(x) &= I(x) \\ I_{\vec{x}}^{\vec{x}_0}(\mathbf{x}) &= \begin{cases} I(\vec{x}_0(i)) , & \text{if } \mathbf{x} \equiv \vec{x}(i) \\ I(\mathbf{x}) , & \text{otherwise} \end{cases} \end{aligned}$$

Because \vec{x}_0 is a permutation of \vec{x} , interval $I_{\vec{x}}^{\vec{x}_0}$ is uniquely defined. The definition satisfies a number of interesting properties, including

$$\left(I_{\vec{x}}^{\vec{x}_0} \right)_{\vec{x}_0}^{\vec{x}} = I$$

and

$$I_0^{\vec{x}_0} =_v I_{\vec{x}}^{\vec{x}_0} \Leftrightarrow I_0 =_v I$$

Corollary 8 (*Properties of renaming program variables in expressions*) Let \vec{x} be a list of unique program variables. Let \vec{x}_0 be a permutation of \vec{x} . If program variables are renamed in an expression, then the following properties hold.

$$\llbracket (e)_{\vec{x}}^{\vec{x}_0} \rrbracket_I = \llbracket e \rrbracket_{I_{\vec{x}}^{\vec{x}_0}}$$

Structural induction over expression e .

- Case $e \equiv X$: Property trivially holds, because $(X)_{\vec{x}}^{\vec{x}_0} \equiv X$ and $I_{\vec{x}}^{\vec{x}_0}(X) = I(X)$.
- Case $e \equiv w$: If $w \notin \vec{x}$, then $(w)_{\vec{x}}^{\vec{x}_0} \equiv w$ and $I_{\vec{x}}^{\vec{x}_0}(w) = I(w)$. Thus, property trivially holds. Therefore assume $w \in \vec{x}$. If $w \equiv \vec{x}(i)$, then

$$(w)_{\vec{x}}^{\vec{x}_0} \equiv \vec{x}_0(i)$$

The way $I_{\vec{x}}^{\vec{x}_0}$ has been constructed ensures

$$\llbracket \vec{x}(i) \rrbracket_{I_{\vec{x}}^{\vec{x}_0}} = I_{\vec{x}}^{\vec{x}_0}(\vec{x}(i))(0) = I(\vec{x}_0(i))(0) = \llbracket \vec{x}_0(i) \rrbracket_I$$

- Case $e \equiv w'$: similar to $e \equiv w$.
- Case $e \equiv w''$: similar to $e \equiv w$.

- Case $e \equiv f(e_1, \dots, e_n)$: directly follows from semantics of function application and induction hypothesis.
- Case $e \equiv (e_1 = e_2)$: directly follows from semantics of equations and induction hypothesis.
- Case $e \equiv (\exists v. \varphi)$:

$$\begin{aligned}
& I \models (\exists v. \varphi)_{\vec{x}}^{\vec{x}_0} \\
\Leftrightarrow & I \models \exists v. (\varphi)_{\vec{x}}^{\vec{x}_0} && \text{Def. 35} \\
\Leftrightarrow & \text{there exists } I_0 \text{ with } I_0 =_x I \text{ and } I_0 \models (\varphi)_{\vec{x}}^{\vec{x}_0} && \text{Def. } \exists \\
\Leftrightarrow & \text{there exists } I_0 \text{ with } I_0 =_x I \text{ and } I_0^{\vec{x}_0} \models \varphi && \text{I.H.} \\
\Leftrightarrow & \text{there exists } I_0 \text{ with } I_0^{\vec{x}_0} =_x I_{\vec{x}}^{\vec{x}_0} \text{ and } I_0^{\vec{x}_0} \models \varphi && \text{see above} \\
\Leftrightarrow & \text{there exists } I_1 \text{ with } I_1 =_x I_{\vec{x}}^{\vec{x}_0} \text{ and } I_1 \models \varphi && (*) \\
\Leftrightarrow & I_{\vec{x}}^{\vec{x}_0} \models \exists v. \varphi && \text{Def. } \exists
\end{aligned}$$

The equivalence (*) can be established, because of $(I_1^{\vec{x}_0})_{\vec{x}}^{\vec{x}_0} = I_1$ (see above).

- Case $e \equiv l_1 :: \varphi_1 \parallel^< l_2 :: \varphi_2$:

$$\begin{aligned}
& I \models (l_1 :: \varphi_1 \parallel^< l_2 :: \varphi_2)_{\vec{x}}^{\vec{x}_0} \\
\Leftrightarrow & I \models (l_1)_{\vec{x}}^{\vec{x}_0} :: (\varphi_1)_{\vec{x}}^{\vec{x}_0} \parallel^< (l_2)_{\vec{x}}^{\vec{x}_0} :: (\varphi_2)_{\vec{x}}^{\vec{x}_0} \\
\Leftrightarrow & \text{there exist } I_1, I_2, n \in \mathbb{N}_0 \\
& \text{with } I \in \llbracket (l_1)_{\vec{x}}^{\vec{x}_0} :: I_1 < \parallel_{n+1} (l_2)_{\vec{x}}^{\vec{x}_0} :: I_2 \rrbracket \\
& \text{and } I_1 \models (\varphi_1)_{\vec{x}}^{\vec{x}_0} \text{ and } I_2 \models (\varphi_2)_{\vec{x}}^{\vec{x}_0} \\
\Leftrightarrow & \text{there exist } I_1, I_2, n \in \mathbb{N}_0 \\
& \text{with } I \in \llbracket (l_1)_{\vec{x}}^{\vec{x}_0} :: I_1 < \parallel_{n+1} (l_2)_{\vec{x}}^{\vec{x}_0} :: I_2 \rrbracket \\
& \text{and } I_1^{\vec{x}_0} \models \varphi_1 \text{ and } I_2^{\vec{x}_0} \models \varphi_2
\end{aligned}$$

Construction of $I_1^{\vec{x}_0}$ and $I_2^{\vec{x}_0}$ ensure that

$$\begin{aligned}
& I \in \llbracket (l_1)_{\vec{x}}^{\vec{x}_0} :: I_1 < \parallel_{n+1} (l_2)_{\vec{x}}^{\vec{x}_0} :: I_2 \rrbracket \\
\Leftrightarrow & I_{\vec{x}}^{\vec{x}_0} \in \llbracket l_1 :: I_1^{\vec{x}_0} < \parallel_{n+1} l_2 :: I_2^{\vec{x}_0} \rrbracket
\end{aligned}$$

Thus

$$\begin{aligned}
& \text{there exist } I_1, I_2, n \in \mathbb{N}_0 \\
& \text{with } I \in \llbracket (l_1)_{\vec{x}}^{\vec{x}_0} :: I_1 < \parallel_{n+1} (l_2)_{\vec{x}}^{\vec{x}_0} :: I_2 \rrbracket \\
& \text{and } I_1^{\vec{x}_0} \models \varphi_1 \text{ and } I_2^{\vec{x}_0} \models \varphi_2 \\
\Leftrightarrow & \text{there exist } I_1, I_2, n \in \mathbb{N}_0 \\
& \text{with } I_{\vec{x}}^{\vec{x}_0} \in \llbracket l_1 :: I_1^{\vec{x}_0} < \parallel_{n+1} l_2 :: I_2^{\vec{x}_0} \rrbracket \\
& \text{and } I_1^{\vec{x}_0} \models \varphi_1 \text{ and } I_2^{\vec{x}_0} \models \varphi_2
\end{aligned}$$

□

(Proof for Lem. 4)

Case “ \Rightarrow ”: Assume $\models \varphi$. Show $\models (\varphi)_{\bar{x}}^{\bar{x}_0}$.

$$\begin{aligned} & \models (\varphi)_{\bar{x}}^{\bar{x}_0} \\ \Leftrightarrow & \text{ for all } I, I \models (\varphi)_{\bar{x}}^{\bar{x}_0} \\ \Leftrightarrow & \text{ for all } I, \llbracket (\varphi)_{\bar{x}}^{\bar{x}_0} \rrbracket_I = \text{tt} \end{aligned}$$

Assume that there is I with $\llbracket (\varphi)_{\bar{x}}^{\bar{x}_0} \rrbracket_I = \text{ff}$. With Cor. 8, it follows that there is I_0 with $\llbracket \varphi \rrbracket_{I_0} = \text{ff}$. Thus, $\models \varphi$ does not hold.

Case “ \Leftarrow ”: similar to “ \Rightarrow ”.

□

C.2 Execution

C.2.1 Lemma 17 (Execution of sequential composition)

Rule *comp lem*

The rule follows from the semantics of $[]$ (see Def. 18) and $;$ (see Def. 13). □

Rule *comp dis*

Rule can be directly derived from the semantics of $;$. □

rule *comp ex*

Relies on the coincidence lemma (see Thm. 1) and refers to the semantics of \exists (see Def. 12). □

Rule *comp lst*

Following from the semantics of sequential composition, we receive for the left hand side

$$\begin{aligned} & I \models (\rho \wedge \mathbf{last}); \psi \\ \Leftrightarrow & \text{ there exists } n \leq |I| \text{ with } I|_n \models \rho \wedge \mathbf{last} \text{ and } I|_n \models \psi \\ & \text{ or } |I| = \infty \text{ and } I \models \rho \wedge \mathbf{last} \end{aligned}$$

The second case contradicts the semantics of **last** (an infinite interval does not satisfy **last**). In the first case, we use $n = 0$ to receive

$$\begin{aligned} & I|_0 \models \rho \wedge \mathbf{last} \text{ and } I|_0 \models \psi \\ \Leftrightarrow & (I(0)) \models \rho \text{ and } (I(0)) \models \mathbf{last} \text{ and } I \models \psi \end{aligned}$$

$(I(0)) \models \mathbf{last}$ trivially holds. Furthermore, with Lemma 9.3, we receive $(I(0)) \models \rho^{[w,w/w',w'']}$. As $\rho^{[w,w/w',w'']}$ is a dynamic expression which depends on the first state of an interval alone, we can apply Lemma 3 to receive $I \models \rho^{[w,w/w',w'']}$. Finally,

$$\begin{aligned} & I \models \rho^{[w,w/w',w'']} \text{ and } I \models \psi \\ \Leftrightarrow & I \models \rho^{[w,w/w',w'']} \wedge \psi \end{aligned}$$

□

Rule *comp stp*

Following from the semantics of sequential composition, we receive for the left hand side

$$\begin{aligned} & I \models (\tau^c \wedge \circ \varphi); \psi \\ \Leftrightarrow & \text{there exists } n \leq |I| \text{ with } I|_n \models \tau^c \wedge \circ \varphi \text{ and } I|_n \models \psi \\ & \text{or } |I| = \infty \text{ and } I \models \tau^c \wedge \circ \varphi \end{aligned}$$

Here, we only consider the first case. A proof for the second case is similar. Expanding the definitions of \wedge and \circ gives

$$\begin{aligned} & I|_n \models \tau^c \wedge \circ \varphi \text{ and } I|_n \models \psi \\ \Leftrightarrow & I|_n \models \tau^c \text{ and } n \geq 1 \text{ and } I|_1^n \models \varphi \text{ and } I|_n \models \psi \quad \text{Sem. } \wedge, \circ \end{aligned}$$

The right hand side expands to

$$\begin{aligned} & I \models \tau^c \wedge \circ (\varphi; \psi) \\ & I \models \tau^c \text{ and } |I| \geq 1 \text{ and } I|_1 \models \varphi; \psi \quad \text{Sem. } \wedge, \circ \end{aligned}$$

For both sides, $|I| \geq 1$. With Lemma 9, it follows that $I \models \tau^c \Leftrightarrow I|_1 \models \tau^c \Leftrightarrow I|_n \models \tau^c$. Continuing with $I|_1 \models \varphi; \psi$, we receive

$$\begin{aligned} & I|_1 \models \varphi; \psi \\ \Leftrightarrow & \text{there exists } m \leq |I| - 1 \text{ with } (I|_1)|^m \models \varphi \text{ and } (I|_1)|_m \models \psi \\ & \text{or } |I| = \infty \text{ and } I|_1 \models \varphi \end{aligned}$$

The first case of the left hand side corresponds to the first case of the right hand side. Take $m = n - 1$ to receive

$$\begin{aligned} & (I|_1)|^{n-1} \models \varphi \text{ and } (I|_1)|_{n-1} \models \psi \\ \Leftrightarrow & I|_1^{n-1+1} \models \varphi \text{ and } I|_{n-1+1} \models \psi \\ \Leftrightarrow & I|_1^n \models \varphi \text{ and } I|_n \models \psi \end{aligned}$$

Thus, the equivalence of both sides has been established. □

C.3 Induction

C.3.1 Theorem 3 (Induction)

Rule $ind(\tau)$

We must show

$$\frac{\vdash N = \tau \wedge \bullet \square (\tau < N \rightarrow \mathbf{ih}) \rightarrow \mathbf{ih}}{\vdash \mathbf{ih}} \quad ind(\tau) .$$

The conclusion must be satisfied by all intervals I , i.e., for an arbitrary I , we must show that

$$I \models \mathbf{ih} . \quad (C.6)$$

With noetherian induction over $\llbracket \tau \rrbracket_I$, we can assume

$$I_0 \models \mathbf{ih} \text{ for all } I_0 \text{ with } \llbracket \tau \rrbracket_{I_0} < \llbracket \tau \rrbracket_I \quad (C.7)$$

as induction hypothesis. Furthermore, the premise of $ind(\tau)$ can be assumed which reads

$$I_0 \models N = \tau \wedge \bullet \square (\tau < N \rightarrow \mathbf{ih}) \rightarrow \mathbf{ih} \text{ for all } I_0 .$$

Semantics of \rightarrow and \wedge give

$$I_0 \models N = \tau \text{ and } I_0 \models \bullet \square (\tau < N \rightarrow \mathbf{ih}) \Rightarrow I_0 \models \mathbf{ih} \text{ for all } I_0 .$$

Take $I_1 := I[\llbracket \tau \rrbracket_{I/N}]$ to receive

$$I_1 \models N = \tau \text{ and } I_1 \models \bullet \square (\tau < N \rightarrow \mathbf{ih}) \Rightarrow I_1 \models \mathbf{ih} \quad (C.8)$$

The first precondition of (C.8) holds, because

$$\begin{aligned} & I_1 \models N = \tau \\ \Leftrightarrow & \llbracket N \rrbracket_{I_1} = \llbracket \tau \rrbracket_{I_1} \quad \text{Sem. } = \\ \Leftrightarrow & \llbracket \tau \rrbracket_I = \llbracket \tau \rrbracket_{I_1} \quad \text{Def. } I_1 \\ \Leftrightarrow & \llbracket \tau \rrbracket_I = \llbracket \tau \rrbracket_I \quad \text{Lem. 2, } n \notin \text{free}(\tau) . \end{aligned}$$

The second precondition of (C.8) can be established as follows.

$$\begin{aligned} & I_1 \models \bullet \square (\tau < N \rightarrow \mathbf{ih}) \\ \Leftrightarrow & I_1|_1 \models \square (\tau < N \rightarrow \mathbf{ih}) \quad \text{Sem. } \bullet \\ \Leftrightarrow & I_1|i \models \tau < N \rightarrow \mathbf{ih} \text{ for all } i > 0 \quad \text{Sem. } \square \\ \Leftrightarrow & \llbracket \tau \rrbracket_{I_1|i} < \llbracket N \rrbracket_{I_1|i} \Rightarrow I_1|i \models \mathbf{ih} \text{ for all } i > 0 \quad \text{Sem. } \rightarrow, < \\ \Leftrightarrow & \llbracket \tau \rrbracket_{I_1|i} < \llbracket N \rrbracket_{I_1} \Rightarrow I_1|i \models \mathbf{ih} \text{ for all } i > 0 \quad n \text{ is static} \\ \Leftrightarrow & \llbracket \tau \rrbracket_{I_1|i} < \llbracket \tau \rrbracket_I \Rightarrow I_1|i \models \mathbf{ih} \text{ for all } i > 0 \quad \text{Lem. 2, } n \notin \text{free}(\tau) \end{aligned}$$

For an arbitrary $i > 0$, the last line follows from our induction hypothesis (C.7). Therefore, both preconditions of (C.8) are satisfied and thus

$$I_1 \models \mathbf{ih} .$$

We have defined $I_1 := I[\llbracket \tau \rrbracket_{I/N}]$. Variable $n \notin \text{free}(\mathbf{ih})$ and therefore, Lem. 2 can be used to establish (C.6). \square

Rule *ind app*

This rule can be derived as follows:

$$\frac{\frac{\Gamma \vdash (\tau < N \rightarrow \mathbf{ih}) \rightarrow (\tau < N \rightarrow \text{pl}(\mathbf{ih}))}{\Gamma \vdash (\tau < N \rightarrow \mathbf{ih}) \wedge \bullet \square (\tau < N \rightarrow \mathbf{ih}) \rightarrow (\tau < N \rightarrow \text{pl}(\mathbf{ih}))} \textit{weaken}}{\Gamma \vdash \square (\tau < N \rightarrow \mathbf{ih}) \rightarrow (\tau < N \rightarrow \text{pl}(\mathbf{ih}))} \textit{alw}$$

According to Cor. 4, \mathbf{ih} is equivalent to $\text{pl}(\mathbf{ih}) \vee \text{tl}(\mathbf{ih})$.

$$\Gamma \vdash (\tau < N \rightarrow \text{pl}(\mathbf{ih}) \vee \text{tl}(\mathbf{ih})) \rightarrow (\tau < N \rightarrow \text{pl}(\mathbf{ih}))$$

The precondition $\text{conf}^-(\neg \mathbf{ih}) \subseteq \Gamma$ gives

$$\begin{aligned} & \text{conf}^-(\neg \mathbf{ih}) \subseteq \Gamma \\ \Leftrightarrow & \Gamma \models \bigwedge \text{conf}^-(\neg \mathbf{ih}) && \text{PL} \\ \Leftrightarrow & \Gamma \not\models \bigvee \text{conf}(\mathbf{ih}) && \text{PL} \\ \Leftrightarrow & \Gamma \not\models \text{tl}(\mathbf{ih}) && \text{Cor. 4} \\ \Leftrightarrow & \Gamma \models \text{tl}(\mathbf{ih}) \leftrightarrow \text{false} && \text{PL} \end{aligned}$$

Thus,

$$\frac{\Gamma \vdash (\tau < N \rightarrow \text{pl}(\mathbf{ih})) \rightarrow (\tau < N \rightarrow \text{pl}(\mathbf{ih}))}{\Gamma \vdash (\tau < N \rightarrow \text{pl}(\mathbf{ih}) \vee \text{false}) \rightarrow (\tau < N \rightarrow \text{pl}(\mathbf{ih}))} \begin{array}{l} \textit{ax} \\ \textit{simp} \end{array}$$

□

Rule *ind weaken*

This rule can easily be derived from *alw* and *weaken*.

C.3.2 Lemma 28 (Induction with liveness)**Rule** *ev cnt*

$$\begin{aligned} & I \models \diamond \varphi \leftrightarrow \exists n. n = n'' + 1 \text{ \textbf{until} } \varphi \\ \Leftrightarrow & I \models \diamond \varphi \Leftrightarrow I \models \exists n. n = n'' + 1 \text{ \textbf{until} } \varphi && \text{Sem. } \leftrightarrow \\ \Leftrightarrow & \text{there exists } i \text{ with } I|_i \models \varphi && \text{Sem. } \diamond, \exists \\ \Leftrightarrow & \text{there exists } I_0 =_n I \text{ with } I_0 \models n = n'' + 1 \text{ \textbf{until} } \varphi \end{aligned}$$

“ \Rightarrow ” For an arbitrary i , $I|_i \models \varphi$. Take $I_0 := I^{(i, i-1, \dots, 0, \dots)/n}$. Thus $I_0 =_n I$. Variable n is fresh, i.e., $n \notin \text{free}(\varphi)$. Therefore, $I_0|_i \models \varphi$. Furthermore, $I_0|_j \models n = n'' + 1$ for all $j < i$. Thus $I_0 \models n = n'' + 1 \text{ \textbf{until} } \varphi$.

“ \Leftarrow ” For an arbitrary I_0 with $I_0 =_n I$, $I_0 \models n = n'' + 1 \text{ \textbf{until} } \varphi$. Trivially, also $I_0 \models \diamond \varphi$. Because $n \notin \text{free}(\varphi)$, $I \models \diamond \varphi$.

□

C.3.3 Lemma 29 (Liveness and LTL operators)**Rule *ev live***

In order to prove *ev live*, we show that

$$I \models \diamond \varphi \Leftrightarrow I \models \exists l. \diamond l \wedge (\neg l) \textbf{ until } \varphi \quad (\text{C.10})$$

Expanding the left hand side gives

$$\begin{aligned} & I \models \diamond \varphi \\ \Leftrightarrow & \text{there exists } i \text{ with } I|_i \models \varphi \quad \text{Sem. } \diamond . \end{aligned}$$

Expanding the right hand side leads to

$$\begin{aligned} & I \models \exists l. \diamond l \wedge (\neg l) \textbf{ until } \varphi \\ \Leftrightarrow & \text{there exists } I_0 \quad \text{Sem. } \exists, \wedge \\ & \text{with } I_0 =_l I \text{ and } I_0 \models \diamond l \text{ and } I_0 \models (\neg l) \textbf{ until } \varphi \\ \Leftrightarrow & \text{there exists } I_0, i, j \quad \text{Sem. } \diamond, \textbf{ until } . \\ & \text{with } I_0 =_l I \\ & \text{and } I_0|_i \models l \\ & \text{and } I_0|_j \models \varphi \text{ and } I_0|_k \models \neg l \text{ for all } k < j \end{aligned}$$

Proof of (C.10):

“ \Leftarrow ” For an arbitrary I_0 there exists an index j with $I_0|_j \models \varphi$. Because $I_0 =_l I$ and l does not freely occur in φ , also $I|_j \models \varphi$.

“ \Rightarrow ” For an arbitrary index i with $I|_i \models \varphi$ construct I_0 with $I_0 =_l I$ and $I_0|_i \models l$ and $I_0|_k \models \neg l$ for all $k < i$. Because $I|_i \models \varphi$, and $l \notin \text{free}(\varphi)$, we can derive for $i = j$, $I_0|_j \models \varphi$.

□

Rule *until live*

Proof is similar to rule *ev live*.

Rule *alw safe*

In order to prove *alw safe*, we show that

$$I \models \square \varphi \Leftrightarrow I \models \forall l. \diamond l \rightarrow \varphi \textbf{ unless } l \quad (\text{C.11})$$

Expanding the left hand side gives

$$\begin{aligned} & I \models \square \varphi \\ \Leftrightarrow & I|_i \models \varphi \text{ for all } i \quad \text{Sem. } \square . \end{aligned}$$

Expanding the right hand side leads to

$$\begin{aligned}
& I \models \forall l. \diamond l \rightarrow \varphi \textbf{ unless } l \\
\Leftrightarrow & \text{ for all } I_0, && \text{Sem. } \exists, \rightarrow \\
& \quad I_0 =_l I \text{ and } I_0 \models \diamond l \Rightarrow I_0 \models \varphi \textbf{ unless } l \\
\Leftrightarrow & \text{ for all } I_0, i, j, && \text{Sem. } \diamond, \textbf{ unless } . \\
& \quad I_0 =_l I \\
& \quad \text{and } I_0|_i \models l \\
& \Rightarrow I_0|_j \models \varphi \text{ or there exists } k \leq j \text{ with } I_0|_k \models l
\end{aligned}$$

Proof of (C.11):

“ \Rightarrow ” Take an arbitrary I_0, j with $I_0 =_l I$. The left hand side ensures that $I|_j \models \varphi$. Because l does not freely occur in φ , also $I_0|_j \models \varphi$.

“ \Leftarrow ” For an arbitrary index i , it must be shown that $I|_i \models \varphi$. Take I_0 with $I_0 =_l I$ and $I_0|_{i+1} \models l$ and $I_0|_k \models \neg l$ for all $k \leq i$. With $i = j$ it follows from the right hand side that $I_0|_i \models \varphi$. l does not freely occur in φ and therefore $I|_i \models \varphi$.

□

Rule *unls safe*

Proof is similar to rule *alw safe*.

C.3.4 Lemma 30 (Liveness and chop)

Rule *chp live*

Property *chp live* can be refined to some extent with existing calculus rules.

$$\begin{array}{c}
\frac{\diamond l_1; \text{true} \vdash \diamond l_1}{(\diamond l_1 \wedge \text{true}); \text{true} \vdash \diamond l_1} \text{simp} \quad \frac{}{(\text{true} \wedge \varphi_1); \psi \vdash \varphi_1; \psi} \text{simp} \\
\frac{}{(\diamond l_1 \wedge \varphi_1); \psi \vdash \diamond l_1} \text{weaken} \quad \frac{}{(\diamond l_1 \wedge \varphi_1); \psi \vdash \varphi_1; \psi} \text{weaken} \\
\hline
\frac{(\diamond l_1 \wedge \varphi_1); \psi \vdash \diamond l_1 \wedge \varphi_1; \psi}{(\diamond l_1 \wedge \varphi_1); \psi \vdash \exists l_0. \diamond l_0 \wedge \varphi_0; \psi} \text{ex } r(l_1) \\
\frac{}{\exists l_0. (\diamond l_0 \wedge \varphi_0); \psi \vdash \exists l_0. \diamond l_0 \wedge \varphi_0; \psi} \text{simp} \\
\frac{}{(\exists l. \diamond l \wedge \varphi); \psi \vdash \exists l_0. \diamond l_0 \wedge \varphi_0; \psi} \text{chp ex} \\
\hline
\frac{}{\vdash (\exists l. \diamond l \wedge \varphi); \psi \rightarrow \exists l_0. \diamond l_0 \wedge \varphi_0; \psi} \text{simp}
\end{array}$$

where $\varphi_1 \equiv \varphi^{[l_1/i]}$. The only remaining premise is established on meta level.

$$\begin{aligned}
& I \models \diamond l_1; \text{true} \rightarrow \diamond l_1 \\
\Leftrightarrow & I \models \diamond l_1; \text{true} \Rightarrow I \models \diamond l_1 \quad \text{Sem. } \rightarrow
\end{aligned}$$

It must be shown that $I \models \diamond l_1$. Refining the left hand side

$$\begin{aligned} & I \models \diamond l_1; \text{true} \\ \Leftrightarrow & \quad \text{there exists } i \leq |I| \text{ with } I|_i \models \diamond l_1 \text{ and } I|_i \models \text{true} \quad \text{Sem. ;} \\ & \text{or } |I| = \infty \text{ and } I \models \diamond l_1 \end{aligned}$$

gives two cases. The second case is trivial, as $I \models \diamond l_1$. For the first case, we reason as follows.

$$\begin{aligned} & \text{there exists } i \leq |I| \text{ with } I|_i \models \diamond l_1 \text{ and } I|_i \models \text{true} \\ \Leftrightarrow & \text{there exists } i \leq |I| \text{ with } I|_i \models \diamond l_1 \quad \text{Logic} \\ \Leftrightarrow & \text{there exists } j \leq i \leq |I| \text{ with } I|_j \models l_1 \quad \text{Sem. } \diamond \\ \Leftrightarrow & \text{there exists } j \leq i \leq |I| \text{ with } I(j)(l_1) = \text{tt} \quad \text{Sem. variable} \end{aligned}$$

As can be seen, the upper bound i of the interval is not relevant for the evaluation of state formula l_1 . Therefore, we are able to lift the property from the prefix interval $I|_i$ to the complete interval I .

$$\begin{aligned} & \text{there exists } j \leq i \leq |I| \text{ with } I(j)(l_1) = \text{tt} \\ \Leftrightarrow & \text{there exists } j \leq |I| \text{ with } I(j)(l_1) = \text{tt} \quad \text{Logic} \\ \Leftrightarrow & \text{there exists } j \leq |I| \text{ with } I|_j \models l_1 \quad \text{Sem. variable} \\ \Leftrightarrow & I \models \diamond l_1 \quad \text{Sem. } \diamond \end{aligned}$$

Thus, the right hand side is also established for the first case.. □

C.3.5 Lemma 31 (Liveness and interleaving)

Rule *ilv live 1*

W.l.o.g., assume $l \notin \text{free}(l_1, l_2, \psi) \cup (\text{free}(\varphi) \setminus \{l\})$. Prove

$$\begin{aligned} & I \models l_1 :: (\exists l. \diamond l \wedge \varphi) \parallel l_2 :: \psi \\ \Rightarrow & I \models \exists l. \diamond l \wedge (l \vee l_1) :: \varphi \parallel l_2 :: \psi \end{aligned}$$

Expanding the left hand side gives

$$\begin{aligned} & I \models l_1 :: (\exists l. \diamond l \wedge \varphi) \parallel l_2 :: \psi \\ \Leftrightarrow & \quad I \models l_1 :: (\exists l. \diamond l \wedge \varphi) \parallel^< l_2 :: \psi \quad \text{Def. } \parallel \\ & \text{or } I \models l_2 :: \psi \parallel^< l_1 :: (\exists l. \diamond l \wedge \varphi) \end{aligned}$$

The two cases are very similar. We therefore concentrate on the first

$$I \models l_1 :: (\exists l. \diamond l \wedge \varphi) \parallel^< l_2 :: \psi .$$

According to the declarative semantics of interleaving, there exist intervals I_1, I_2 with $I_1 \models \exists l. \diamond l \wedge \varphi$ and $I_2 \models \psi$ and a fair scheduling sequence $(p_i)_{i=0}^{|I|}$ with the properties

of Lemma 1.

$$\begin{aligned} & I_1 \models \exists l. \diamond l \wedge \varphi \\ \Leftrightarrow & \text{there exists } I_{1,0} \text{ with } \begin{array}{l} I_{1,0} =_l I_1 \\ \text{and } I_{1,0} \models \diamond l \\ \text{and } I_{1,0} \models \varphi \end{array} \quad \text{Sem. } \exists, \wedge \end{aligned}$$

Expanding the right hand side gives

$$\begin{aligned} & I \models \exists l. \diamond l \wedge (l \vee l_1) :: \varphi \parallel l_2 :: \psi \\ \Leftrightarrow & \text{there exists } I_0 \text{ with } \begin{array}{l} I_0 =_l I \\ \text{and } I_0 \models \diamond l \\ \text{and } I_0 \models (l \vee l_1) :: \varphi \parallel l_2 :: \psi \end{array} \quad \text{Sem. } \exists, \wedge \end{aligned}$$

Take I_0 with $I_0 =_l I$ and

$$I_0(i)(l) = \begin{cases} I_{1,0}(pc_1^{(i)})(l), & \text{if } 1 \in \pi_i \\ \text{ff}, & \text{otherwise} \end{cases}$$

The definition of $I_0(i)(l)$ takes the value of l in $I_{1,0}$, if the first process is scheduled for execution in step i . Otherwise l evaluates to ff. The primed value $I_0(i)'(l)$ is arbitrary and will be determined by the interleaving below.

$I_0 \models \diamond l$ holds, because

$$\begin{aligned} & \text{there exists } i \text{ with } I_{1,0}(i)(l) = \text{tt} \\ \Leftrightarrow & \text{there exists } i \text{ with } I_{1,0}(pc_1^{((pc_1^{-1})^{(i)})})(l) = \text{tt} \\ \Leftrightarrow & \text{there exists } i \text{ with } I_0((pc_1^{-1})^{(i)})(l) = \text{tt} \quad \text{Def. } I_0 \end{aligned}$$

The latter is because the scheduling sequence (p_i) is fair.

It remains to prove:

$$I_0 \models (l \vee l_1) :: \varphi \parallel l_2 :: \psi$$

According to the declarative semantics of interleaving, we have to find intervals $I_{0,1}, I_{0,2}$ with $I_{0,1} \models \varphi$ and $I_{0,2} \models \psi$ and a fair scheduling sequence $(p_{0,i})_{i=0}^{|I|}$ with the properties of Lemma 1. Take

$$\begin{aligned} I_{0,1} & := I_{1,0}, \\ I_{0,2} & =_l I_2, \\ I_{0,2}(i)(l) & := I_0((pc_2^{-1})^{(i)})(l) \\ (p_{0,i}) & := (p_i). \end{aligned}$$

Thus, $p_{0,0} = p_0 = 1$, and $|I_0| = |I| = \infty$ or $p_{0,|I|} = p_{|I|} = \emptyset$. Because $I_{0,1} = I_{1,0} =_l I_1$ and $I_{0,2} =_l I_2$, the derived set of actual schedules (π_i) is unchanged. Conditions 1 - 6 of Def. 1 can also be verified.

□

Rule *ilv live 2*

Similar to rule *ilv live 1*.

C.3.6 Lemma 32 (Repeated induction)**Rule *rep ind***

The until formula guarantees that $\diamond \varphi$ is reached. Instead of property *until live* to extract liveness, a different property is more useful in this case.

$$\text{until live}': \quad \varphi \text{ until } \psi \quad \leftrightarrow \quad \varphi \text{ until } \psi \wedge \diamond \psi$$

Starting with the conclusion, standard induction with rule *ind ev* can be applied.

$$\frac{\frac{\frac{\frac{\vdash \bullet \mathbf{ih}_0 \text{ until } \varphi \wedge \bullet \mathbf{ih} \text{ until } \varphi \rightarrow \psi}{\vdash \bullet \mathbf{ih} \text{ until } \varphi \rightarrow (\bullet \mathbf{ih}_0 \text{ until } \varphi \rightarrow \psi)} \text{pl}}{\vdash \diamond \varphi \rightarrow (\bullet \mathbf{ih}_0 \text{ until } \varphi \rightarrow \psi)} \text{pl}}{\vdash \bullet \mathbf{ih}_0 \text{ until } \varphi \wedge \diamond \varphi \rightarrow \psi} \text{pl}}{\vdash \bullet \mathbf{ih}_0 \text{ until } \varphi \rightarrow \psi} \text{until live}'$$

where $\mathbf{ih} \equiv \bullet \mathbf{ih}_0 \text{ until } \varphi \rightarrow \psi$. It remains to unite and simplify the two induction formulas to receive the premise of rule *rep ind*.

Some additional properties of operator **until** and \bullet are required.

$$\begin{aligned} \text{until cntret:} \quad & \varphi_1 \text{ until } \psi \wedge \varphi_2 \text{ until } \psi \\ & \leftrightarrow (\varphi_1 \wedge \varphi_2 \wedge \varphi_1 \text{ until } \psi) \text{ until } \psi \\ \text{wnx cntret:} \quad & \bullet \varphi \wedge \bullet \psi \quad \leftrightarrow \quad \bullet (\varphi \wedge \psi) \\ \text{wnx snx cntret:} \quad & \bullet \varphi \wedge \circ \psi \quad \leftrightarrow \quad \circ (\varphi \wedge \psi) \\ \text{snx wnx:} \quad & \circ \varphi \quad \rightarrow \quad \bullet \varphi \end{aligned}$$

Property *until cntret* is especially designed for the current proof. If both properties φ_1 and φ_2 hold until ψ , it is possible to contract the two **until** formulas. In addition, one of the formulas can be added to the first formula of the resulting **until** as additional context. The other properties are very simple.

Proof continues as follows.

$$\frac{\frac{\frac{\vdash (\bullet \mathbf{ih}_0 \wedge \bullet \mathbf{ih} \wedge \circ (\bullet \mathbf{ih}_0 \text{ until } \varphi)) \text{ until } \varphi \wedge \bullet \mathbf{ih} \text{ until } \varphi \rightarrow \psi}{\vdash (\bullet \mathbf{ih}_0 \wedge \bullet \mathbf{ih} \wedge (\text{false} \vee \circ (\bullet \mathbf{ih}_0 \text{ until } \varphi))) \text{ until } \varphi \wedge \bullet \mathbf{ih} \text{ until } \varphi \rightarrow \psi} \text{simp}}{\vdash (\bullet \mathbf{ih}_0 \wedge \bullet \mathbf{ih} \wedge (\varphi \vee \circ (\bullet \mathbf{ih}_0 \text{ until } \varphi))) \text{ until } \varphi \wedge \bullet \mathbf{ih} \text{ until } \varphi \rightarrow \psi} \text{simp}}{\vdash (\bullet \mathbf{ih}_0 \wedge \bullet \mathbf{ih} \wedge \bullet \mathbf{ih}_0 \text{ until } \varphi) \text{ until } \varphi \wedge \bullet \mathbf{ih} \text{ until } \varphi \rightarrow \psi} \text{until}}{\vdash \bullet \mathbf{ih}_0 \text{ until } \varphi \wedge \bullet \mathbf{ih} \text{ until } \varphi \rightarrow \psi} \text{until cntret}$$

We continue with contracting the \bullet and \circ formulas.

$$\frac{\frac{\frac{\vdash \bullet (\mathbf{ih}_0 \wedge \mathbf{ih} \wedge \bullet \mathbf{ih}_0 \text{ until } \varphi) \text{ until } \varphi \wedge \bullet \mathbf{ih} \text{ until } \varphi \rightarrow \psi}{\vdash \circ (\mathbf{ih}_0 \wedge \mathbf{ih} \wedge \bullet \mathbf{ih}_0 \text{ until } \varphi) \text{ until } \varphi \wedge \bullet \mathbf{ih} \text{ until } \varphi \rightarrow \psi} \text{ snx wnx}}{\vdash (\bullet (\mathbf{ih}_0 \wedge \mathbf{ih}) \wedge \circ (\bullet \mathbf{ih}_0 \text{ until } \varphi)) \text{ until } \varphi \wedge \bullet \mathbf{ih} \text{ until } \varphi \rightarrow \psi} \text{ wnx snx cntret}}{\vdash (\bullet \mathbf{ih}_0 \wedge \bullet \mathbf{ih} \wedge \circ (\bullet \mathbf{ih}_0 \text{ until } \varphi)) \text{ until } \varphi \wedge \bullet \mathbf{ih} \text{ until } \varphi \rightarrow \psi} \text{ wnx cntret}$$

Both induction formulas have been contracted. Sub formula

$$\mathbf{ih}_0 \wedge \mathbf{ih} \wedge \bullet \mathbf{ih}_0 \text{ until } \varphi$$

can be further simplified. Expanding \mathbf{ih} gives

$$\mathbf{ih}_0 \wedge (\bullet \mathbf{ih}_0 \text{ until } \varphi \rightarrow \psi) \wedge \bullet \mathbf{ih}_0 \text{ until } \varphi$$

The third conjunction can be used as additional context to simplify the second conjunction.

$$\mathbf{ih}_0 \wedge (\text{true} \rightarrow \psi) \wedge \bullet \mathbf{ih}_0 \text{ until } \varphi$$

The formula $\bullet \mathbf{ih}_0 \text{ until } \varphi$ can then be discarded.

$$\mathbf{ih}_0 \wedge (\text{true} \rightarrow \psi) \wedge \text{true}$$

Further simplification leads to

$$\mathbf{ih}_0 \wedge \psi$$

and the final premise reads

$$\text{Inv}, \Gamma, \bullet (\mathbf{ih}_0 \wedge \mathbf{ih}_1) \text{ until } \varphi \vdash \Delta$$

where $\mathbf{ih}_1 \equiv \psi$. □

Appendix D

Mathematical Symbols

A	the set of all algebras	Def. 4	p. 27
D	domain	Def. 4	p. 27
E	the set of all expressions	Def. 2	p. 22
F	the set of all formulas		p. 24
I	the set of all intervals	Def. 6	p. 28
OP	set of operations	Def. 1	p. 21
PROC	set of procedures	Def. 1	p. 21
S	set of sorts	Def. 1	p. 21
SIG	signature	Def. 1	p. 21
X	set of static variables	Def. 1	p. 21
Y	set of dynamic variables	Def. 1	p. 21
Z	set of program variables	Def. 1	p. 21
<i>blk</i>	special boolean program variable		p. 22
<i>d</i>	domain element		
<i>e</i>	expression	Def. 2	p. 22
<i>f</i>	operation	Def. 2	p. 22
<i>i</i>	natural number		
<i>j</i>	natural number		
<i>n</i>	natural number		
\bar{n}	natural number or infinity		
<i>m</i>	natural number		
<i>proc</i>	procedure	Def. 2	p. 22
<i>s</i>	sort	Def. 1	p. 21
<i>t</i>	SOS term		p. 42
<i>X</i>	static variable		p. 24
<i>x</i>	dynamic variable		p. 24
<i>x</i>	program variable		p. 24

I	interval	Def. 6	p. 28
T	sequence of SOS terms		
ε	dynamic expression	Def. 30	p. 49
ε_s	static expression	Def. 30	p. 49
φ	formula		p. 24
ψ	formula		p. 24
σ	state or valuation	Def. 5	p. 27
χ	formula		p. 24
Σ	the set of all states	Def. 5	p. 27
\mathcal{A}	algebra	Def. 4	p. 27

Bibliography

- [1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 3 Nov 1995.
- [2] L. Aceto, W.J. Fokkink, and C. Verhoef. Structural operational semantics. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*. Elsevier, 2001.
- [3] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.
- [4] M. Balsler, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS, pages 363–366. Springer-Verlag, 2000.
- [5] Simon Bäumler. Verification of UML state machines with KIV. Diplomarbeit, Ludwig-Maximilians-Universität München, 2003.
- [6] N. Bjørner, A. Brown, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma, and T. Uribe. Verifying temporal properties of reactive systems: A step tutorial. In *Formal Methods in System Design*, pages 227–270, 2000.
- [7] A. Cau, B. Moszkowski, and H. Zedan. *ITL – Interval Temporal Logic*. Software Technology Research Laboratory, SERCentre, De Montfort University, The Gateway, Leicester LE1 9BH, UK, 2002. www.cms.dmu.ac.uk/~cau/itlhomepage.
- [8] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, number 131 in LNCS. Springer, 1981.
- [9] C. Duelli. Verifikation nebenläufiger Systeme – Werkzeugunterstützung und praktische Erfahrungen. Diplomarbeit, Fakultät für Informatik, Universität Ulm, Germany, 2000. (in German).
- [10] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B, Formal Models and Semantics*, pages 995–1072. North-Holland Pub. Co./MIT Press, 1990.

- [11] D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume 2, pages 496–604. Reidel, 1984.
- [12] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
- [13] M. Heisel, W. Reif, and W. Stephan. A Dynamic Logic for Program Verification. In A. Meyer and M. Taitlin, editors, *Logical Foundations of Computer Science*, LNCS 363, pages 134–145, Berlin, 1989. Logic at Botik, Pereslavl-Zalessky, Russia, Springer.
- [14] C.A.R. Hoare. An axiomatic basis for computer programming. *COMM ACM*, pages 576–580, 1969.
- [15] Protocure home page. <http://www.protocure.org>.
- [16] D. Hutter, H. Mantel, G. Rock, W. Stephan, A. Wolpers, M. Balsler, W. Reif, G. Schellhorn, and K. Stenzel. VSE: Controlling the Complexity in Formal Software Developments. In *Current Trends in Applied Formal Methods*, LNCS 1641, Berlin, 1999. Boppard, Germany, Springer.
- [17] Isabelle home page. <http://isabelle.in.tum.de/>.
- [18] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
- [19] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems*. Springer, 1992.
- [20] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems—Safety*. Springer, 1995.
- [21] Z. Manna and A. Pnuelli. Temporal verification diagrams. In M. Hagiya and J. Mitchell, editor, *International Symposium on Theoretical Aspects of Computer Software*, volume 789, pages 726–765. Springer Verlag, 1994.
- [22] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.
- [23] B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, 1986.
- [24] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of LNCS. Springer, 2002.
- [25] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Tutorial on Isabelle/HOL*. Springer LNCS 2283, 2003. available at <http://isabelle.in.tum.de/>.
- [26] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

- [27] W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer-Verlag, Berlin, 1995.
- [28] G. Rock, W. Stephan, and A. Wolpers. Modular reasoning about structured tla specifications. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computer Science, pages 217 – 229. Springer, Wien NewYork, 1998.
- [29] J. Schmitt. Verifikation nebenläufiger Systeme. Diplomarbeit, Universität Augsburg, 2004. (in German).
- [30] H. Sipma. *Diagram-based verification of discrete, real-time and hybrid systems*. PhD thesis, Stanford University, February 1999.
- [31] T. Sorg. Verifikation nebenläufiger Systeme. Diplomarbeit, Fakultät für Informatik, Universität Ulm, Germany, 1999. (in German).
- [32] A. Thums. *Formale Fehlerbaumanalyse*. PhD thesis, Universität Augsburg, Augsburg, Germany, 2004. (in German).

Lebenslauf

Michael Balsler

Persönliche Daten

Geburtsdatum: 05.03.1972
Geburtsort: Ulm
Familienstand: verheiratet
Staatsangehörigkeit: deutsch

Werdegang

09/1978 - 07/1982 Grundschule Burlafingen
09/1982 - 07/1991 Bertha-von-Suttner Gymnasium Neu-Ulm
Abschluss: Allgemeine Hochschulreife
08/1991 - 10/1992 Zivildienst im Mobilen und Sozialen Hilfsdienst
Diakonie Neu-Ulm
10/1992 - 01/1997 Studium der Informatik, Nebenfach Mathematik
Universität Ulm
Abschluss: Dipl. Inf.
02/1997 - 05/2000 Wissenschaftlicher Mitarbeiter
Abteilung Programmiersprachen und Compilerbau
Universität Ulm
06/2000 - 05/2005 Wissenschaftlicher Angestellter
Lehrstuhl Softwaretechnik und Programmiersprachen
Universität Augsburg
07/2005 Abschluss der Promotion