A Dynamic Logic with Temporal Operators for Promela

Florian Rabe

23.10.2004

Abstract

In this work DLTP is introduced as a first-order dynamic logic with trace semantics and temporal operators. The used programming language is a fragment of Promela, which is a simple programming language that allows parallel execution of dynamically created indeterministic processes and asynchronous and synchronous communication between them.

The semantical correspondence between DLTP and Promela arises in the special case where DLTP is used with the standard vocabulary and the standard structure. But many results of this work hold in the general case.

A sequent calculus for the standard structure of DLTP is given and proven to be sound and relatively complete. Finally possible generalizations, strengths, weaknesses and complexity of the logic and the calculus are discussed.

Contents

1 Introduction and Related Work

This work introduces DLTP: a first-order dynamic logic with trace semantics and temporal operators for a fragment of Promela. In this section the terms trace semantics and temporal operators are explained, an introduction to Promela is given and the relation of DLTP to other works is described.

1.1 Temporal Operators in Dynamic Logic

The semantics of dynamic logic is usually defined using Kripke structures (see for example [13]). The states of such a structure are the possible interpretations of the data objects, and each program is interpreted as an accessibility relation between the states. The modal operators $[\pi]$ and $\langle\pi\rangle$ are used to quantify over the final states of the program π.

This interpretation views the execution of a program as a set of pairs of an initial and a final state. It abstracts from the intermediate states of the execution. There is another possibility, which views the execution of a program as a sequence of states that contains all states that are passed during the execution (see for example [14] as well as [21] and [22]).

Such a sequence is called a trace. A trace semantics assigns with each initial state and each program a trace or in the indeterministic case a set (or tree) of traces. This kind of semantics is used in this work.

For a trace semantics the structures still consist of all the possible states, but there are no accessibility relations anymore. Instead a modal operator states a property of the set of traces induced by a program and an initial state.

Of course the usual operators for the quantification over the final states can be defined in this setting. But there are other interesting possibilities, most of which are motivated by branching temporal logics (e. g. CTL as in [9]): Then the initial state is analogous to the present, and the tree of traces induced by a program gives the structure of the future.[1] Simple examples are the quantification "During the execution of the program π, along all traces the formula F eventually holds." or "During the execution of the program π, in one trace the formula F always holds.".

1.2 Spin and Promela

1.2.1 The Spin LTL Model-checking Tool

Spin is an interpreter, simulator and verifier for the programming language Promela. Promela is a programming language that is used to describe the indeterministic interaction of indeterministic processes. Spin and Promela have been developed together at the Bell Labs in 1980 and have been freely available since 1991.

The Spin version that is used in this work is given by [2]. The Promela specification (as part of the Spin documentation) is given in [1].

The main features of Spin are[2]:

- random or interactive simulations
- exhaustive verification
- checking for deadlocks and not executable commands
- proving system invariants, finding non-progress execution cycles and verifying linear time temporal constraints

Spin does not offer deductive verification of statements about Promela programs; that is what DLTP allows.

Promela is not only a programming language, but contains additional syntactical elements that are used to formulate constraints and claims that are evaluated by Spin. It also contains syntactical elements to improve the efficiency of the verification. Since DLTP is developed to verify Promela programs logically all these syntactical elements are omitted (see section 9.2) and Spin is simply regarded as an interpreter of Promela.

In section 9 there are some remarks on minor differences between the Promela specification and the Spin implementation.

1.2.2 The Promela Language

This introduction to Promela is only an abstract survey of the essential properties of the language. A more elaborate description is found in [1].

Programs and States A Promela program essentially consists of the declaration of the process types and the global data objects. A process type declaration consists of the process code and the declaration of its local data objects.

A state is an assignment of values to all global and local data objects of active process instances, a program counter for each process instance and some other information.

When a program is executed the process types that are marked to immediately have an active instance are instantiated, the global and the respective local data objects are initialized, and the program counters of the process instances are set to the first command. Then the next state is iteratively computed until all process instances have terminated.

The next state is computed by indeterministically choosing an elementary command for execution and applying its effect to the current state.

[1]The analogy only extends to those temporal operators that refer only to the future.
[2]See [1] for a comprehensive description.

Data Objects Promela offers the usual integer types and a special data type channel. Also composed data types can be defined in the declaration of a program. Classes and floating point numbers are not provided.

Channels have a type and a capacity and are used as pointers to message queues of objects of the specified type. Channels offer asynchronous or synchronous communication between all process instances by sending to and receiving from a channel. If the capacity is zero the communication must be synchronous. Otherwise the capacity gives the number of objects that may be asynchronously buffered in the queue. A process that creates a channel locally must send it to those process instances that it wants to communicate with. See section 9.3.3 for a discussion of the queue concept.

Executability In Promela there are no run-time errors. Instead a command is only executable in certain states. In every state the executability of all commands at the program counters is checked and only executable commands may be executed.

Elementary Commands The most important elementary commands are:

- Assignments to local or global data objects. Assignments are always executable.
- Expressions. Among the expressions are the usual arithmetical and logical operations. All operations are defined for all number data types and the arguments and the result are cast appropriately.

  The most important special expression is the RUN expression. It has side effects because it creates a new instance of a process type and returns its process ID or zero if an a-priori limit of active process instances has been reached.

  Expression are executable if they are non-zero and they have no effect. They are used to test conditions. In particular creating a new process instance may be not executable. In this work creating a new process instance is a special type of elementary command that is always executable.
- Send and receive commands to access queues. There are two types of send commands that differ in where in the queue (at the end for normal or according to an ordering for sorted sending) the sent object is inserted. Both types are executable if the queue is not full.

  There are four types of receive commands that differ in which object (the first one for normal or the first matching one for random receiving) is received and whether or not the received object is removed from the queue. Receive commands contain a pattern and are only executable if the received element matches the pattern.

  The pattern is a tuple of values or variables. An element in the channel matches the pattern if it agrees with the values of the pattern. For example the pattern $(1, X, 2)$ is matched by $(1, 3, 2)$, but not by $(1, 3, 4)$. Those components of the received element for which there is a variable in the pattern are assigned to the respective variable. In the first example the value 3 would be assigned to X.

  Synchronization is achieved by defining send commands to a synchronous channel to be executable only if there is a matching receive command at the program counter of another process instance. In that case both commands are executed.
- ELSE and BREAK with special meanings as described below.
- GOTO commands that set the program counter of a process instance. These are not considered in this work.

Composed Commands There are several types of composed commands. The most important ones are:

- Selections: A selection has the form

```
IF
  :: option
  ⋮
  :: option
FI
```

where the options are sequences of commands and at most one option may begin with ELSE. The meaning of a selection is that any of the options is indeterministically chosen.

The whole selection is executable if the first command of any option is executable; ELSE is a single command that is always executable and has no effect, but the option beginning with ELSE is only executed if none of the other first commands is executable.

The usual "if $x = 0$ then A else B" is written as

```
IF
  :: x; B
  :: ELSE; A
FI
```

in Promela.

- Repetitions: A repetition is similar to a selection. In the notation IF and FI are replaced with DO and OD. It differs from a selection only in that it is repeated until a BREAK command is encountered.

  The usual "WHILE $x <> 0$ DO A" is written as

  ```
  DO
    :: x; A
    :: ELSE; BREAK
  OD
  ```

  in Promela.

- Atomic sequences: An atomic sequence has the form ATOMIC{body}, where the body is a sequence of commands. The meaning of an atomic sequence is that its execution is not interrupted by other processes (except for some special cases). It is executable if the first command of its body is.

Choosing the Next Command The decision which command is executed is made in two steps:

1. For each active process instance a set of possible next commands is computed.
2. If no process has exclusive control any command is chosen from the union of these sets. If a process has exclusive control one of its commands is chosen if there is one; otherwise it loses exclusive control.

This means that there are two types of indeterminism in Promela: The execution of the active process instances is interleaved indeterministically (indeterministic scheduling). And each process instance is executed indeterministically (indeterministic selection).

To compute the above set of possible next commands of a process instance several cases are distinguished depending on the command at the program counter:

- If it is elementary and executable, this command is returned.
- If the command is a selection or a repetition, the executable first commands of all options are returned.
- If the command is an atomic sequence, the first command of its body is returned if it is executable.

If the returned commands are again composed commands, the algorithm is called recursively.

Applying the Effect Firstly, for each elementary command there is a mapping between states that gives the next state depending on the current state. This effect is not the identity only for assignments, send and receive commands, and for these it is intuitively clear.

Secondly, the effect also includes a change of the program counter of the respective process instance which depends on the composed command from which the elementary is extracted:

- If a command that is not part of a composed command is executed, the program counter of the respective process instance is usually set to the next command. A BREAK command within a repetition sets the program counter to the command after the repetition. A GOTO command specifically sets the program counter.

- If the first command of an option of a selection or repetition is executed, the program counter is set to the second command of this option.
- If the first command of an atomic sequence is executed, the program counter is set to the second command, and the process is set to have exclusive control.
- If the last command of an option of a selection or of the body of an atomic sequence is executed, the program counter is set to the command after the composed command.
- If the last command of an option of a repetition is executed, the program counter is again set to the repetition command.

And thirdly, if a RUN is executed a new process instance is created, its local data objects initialized and its program counter set to the first command. And if the last command of a process instance and all its child processes has been executed the process instance is deconstructed and its local data objects are not part of the state anymore.

1.3 Related Work

Promela Semantics Although much has been written about Spin and Promela there are only a few attempts to formally specify the semantics of Promela, for example [7], [18] and [23]. These have been introduced as preliminary and did not win much recognition (see section 13.2.1). Despite the obvious disadvantages the Spin implementation (e. g. [2]) (or rather its source code) is widely accepted as the official specification.

Therefore this work introduces its own formal semantics of Promela which follows [1] and [2] if possible and resolves open questions by definitions.

Dynamic Logics Several dynamic logics have been introduced (see [15] for a survey). In general there are two variants of dynamic logics. Logics for elementary languages often use atomic programs without specifying the actual language or are extensions of propositional logic, as in [21] and [22], [19] or [20]; these are mainly interesting for computability questions. In [5] a deterministic first-order dynamic logic for a more advanced language is given. But only logics for "real" languages (e. g. Java Card as in [6]) can be used to verify real-life programs.

Promela is somewhat in the middle between these variants: It includes the advanced concepts of parallel non-deterministic computation and dynamic process creation, but is quite elementary in other respects (classes and procedures are omitted for example). For such a language no dynamic logic has been developed so far.

However, in [17] the Temporal Logic of Actions is introduced, which also allows to specify and verify indeterministic concurrent systems. This logic is fundamentally different from DLTP, because it abstracts from the programming language and expresses state changes not by commands, but by first-order formulas that relate two consecutive states with each other. This is elegant for elementary commands, but awkward for composed commands and dynamic object creation. The advantages and disadvantages of this approach are discussed in [17].

Sequents and Relative Completeness Sequents as the elementary objects of a calculus were introduced by Gentzen (see for example [11]).

If a calculus is used to find all formulas that hold in all states of a specific structure, which defines the domain of computation and the elementary operations on these domains, in most cases already the first-order formulas that hold in this structure are not recursively enumerable (This is due to Gödel's incompleteness theorems, see [12].).

If a calculus is complete under the condition that all these first-order formulas are axioms it is called relatively complete. The notion of relative completeness was introduced in [10].

2 Elementary Definitions

Numbers $\mathbb{N}$ and $\mathbb{Z}$ denote the standard models of natural and integer numbers with arithmetical operations and comparison.

Definition 1. *The following abbreviations are used for sets of whole numbers:*

- $\mathbb{N}^* := \mathbb{N} \setminus \{0\}$
- $[m; n] = \{i \in \mathbb{Z} | i \geq m,\ i \leq n\}$ *for* $m, n \in \mathbb{Z}$

The Undefined Symbol The symbol $\perp$ is used on three levels to denote non-definedness:

- on the syntactical level: $\perp^S$ is a constant symbol for the sort S that has no defined value,
- on the semantical level: $\perp$ is the interpretation of $\perp^S$ for all sorts S,
- on the meta-level: $f(x) = \perp$ denotes that the partial mapping f is undefined for x.

DLTP needs to be able to express non-definedness for two reasons: There are some operations that cannot be defined sensibly (e. g. reading from an empty queue or division by zero); and, more importantly, an infinite reservoir of program variables, which are initially undefined, is used to deal with the dynamic creation of processes.

Sequences

Definition 2. *A sequence s over a set A is a mapping from $[1; n]$, for $n \in \mathbb{N}$, or from $\mathbb{N}^*$ to A; $s(i)$ is denoted by s_i. In the first case s is denoted by $(s_1, \ldots, s_n)$ if $n \geq 1$ or by $()$ if $n = 0$.*

For sequences s with domain D and s' the following partial mappings are defined:

- *The length:*

$$\text{Length}(s) = \begin{cases} 0 & \text{if } s = () \\ n & \text{if } s = (s_1, \ldots, s_n) \\ \perp & \text{otherwise} \end{cases}$$

  s is called empty in the first case, finite in the first two cases and infinite in the third case.

  The set of sequences over A of length l is denoted by A^l. Sequences of length 1 are identified with their single element.

- *The sequence from which elements with indices from R are removed:* $\text{Remove}(s, R) = (s_{f(i)})_{i \in D'}$ *where*

$$f(i) = \begin{cases} \min\ M & \text{if } i = 1,\ M = D \setminus R \neq \emptyset \\ \min\ M & \text{if } i > 1,\ M = \big\{j \in D \setminus R | j > f(i-1)\big\} \neq \emptyset \\ \perp & \text{otherwise} \end{cases}$$

  and D' is the domain of f.

- *The sequence where an element a is inserted at a certain position $n \in \mathbb{N}^*$:* $\text{Insert}(s, a, n) = (t_i)_{i \in D'}$ *where* $l = \text{Length}(s)$,

$$D' = \begin{cases} D \cup \{l + 1\} & \text{if } l \in \mathbb{N}^*,\ n \in [1; l+1] \\ D & \text{otherwise} \end{cases}$$

  and

$$t_i = \begin{cases} s_i & \text{if } i < n \\ a & \text{if } i = n \\ s_{i-1} & \text{if } i > n \end{cases}$$

  for $i \in D'$.

- *The first element:* $\text{First}(s) = \begin{cases} s_1 & \text{if Length}(s) \neq 0 \\ \perp & \text{otherwise} \end{cases}$

- *The last element:* $\text{Last}(s) = \begin{cases} s_n & \text{if Length}(s) = n \in \mathbb{N}^* \\ \perp & \text{otherwise} \end{cases}$

- *The rest after the first element is removed:* $\mathrm{Rest}(s) = \mathrm{Remove}(s, \{1\})$
- *The composition:* $s \cdot s' = \begin{cases} (s_1, \ldots, s_n, s'_1, s'_2, \ldots) & \text{if } \mathrm{Length}(s) = n \in \mathbb{N} \\ s & \text{otherwise} \end{cases}$

3 The Logic DLTP

In this section at first vocabularies are defined. Then DLTP is defined as a mapping from vocabularies to instances of the logic. However, the detailed definitions of these instances (syntax, structures, truth values and semantics) are only given in later sections. Finally the standard vocabulary is defined, which is the only vocabulary that is explicitly used in this work.

3.1 Vocabularies

Definition 3. *A* vocabulary *for* DLTP *is a tuple of:*

- *A pair* (Σ_E, Σ_L) *of disjunct sets, called the set of* elementary sorts *and the set of* logical sorts*, respectively.*

  Then the set of sorts is $\Sigma = \Sigma_P \cup \Sigma_L$ *where* Σ_P*, the set of* program sorts *or composed sorts, is the set of finite non-empty sequences over* Σ_E*. The program sort* $(S_1, \ldots, S_n)$ *is denoted by* $S_1 \times \ldots \times S_n$*.*
- *A tuple* $(FS_{RP}, FS_{NP}, FS_{RL}, FS_{NL})$ *of disjunct sets, called the sets of rigid program, non-rigid program, rigid logical and non-rigid logical* function symbols*, respectively.*

  Then the set FS *of function symbols is defined by* $FS = FS_{RP} \cup FS_{NP} \cup FS_{RL} \cup FS_{NL}$*.*
- *A set* $FS_{prim} \subseteq FS_{NP} \cup FS_{NL}$*, called the set of primary non-rigid function symbols.*
- *A set* PS*, called the set of* predicate symbols.
- *The* signature *mapping* Sign *which assigns a finite non-empty sequence of sorts, called a signature, with each function or predicate symbol. The signature* $(S^1, \ldots, S^n)$ *is denoted by* $S^1 \ \ldots \ S^n$*.*

  In that case the arity *of a function symbol is* $n - 1$*, the arity of a predicate symbol is* n*.*

that satisfies the conditions that $\{I, C\} \subseteq \Sigma_E$*,* $\{var^S | S \in \Sigma_P\} \cup \{nextpid\} \subseteq FS_{NP}$*,* $glob \in FS_{RP}$ *and* $\{=^S | S \in \Sigma_P\} \subseteq PS$ *with* $\mathrm{Sign}(var^S) = I\ I\ S$*,* $\mathrm{Sign}(=^S) = S\ S$ *for* $S \in \Sigma_P$ *and* $\mathrm{Sign}(nextpid) = \mathrm{Sign}(glob) = I$*.*

The sorts I and C are numbers and channels, the fundamental types of objects in Promela. var^S gives the program variables of the sort S. The symbols *glob* and *nextpid* are number symbols that appear in the first argument of var^S to distinguish between global and local variables. $=^S$ is equality for the sort $S \in \Sigma_P$. Vocabularies must contain these symbols because they are used in the definition of the syntax.

The notion of primary non-rigid function symbols is introduced in this work. In the standard structure their interpretation characterizes a state, and the non-primary non-rigid function symbols are definable from them. Updates can only be applied to primary non-rigid function symbols. The only reason to use non-primary non-rigid function symbols is that they may occur in programs whereas some of the more expressive primary ones must not.

Every vocabulary for DLTP induces a vocabulary $(\Sigma, FS, PS, \mathrm{Sign})$ for many-sorted first-order predicate logic.

3.2 Definition

Definition 4. *The logic* DLTP *is a tuple of*

- *the syntax: a mapping that assigns a triple consisting of the family of sets of terms, the set of programs and the set of formulas with every vocabulary,*
- *the structure mapping: a mapping that assigns a class of structures with each vocabulary,*
- *the set of truth values: a Boolean lattice,*

- *the semantics: a mapping that assigns certain objets with each pair of a structure and a term, a program or a formula.*

Since only one vocabulary and for that only one structure are considered in this work, a strict set theoretical foundation is avoided. Such a foundation would have to define

- the syntax as a functor from the category of vocabularies to the category of sets,
- the structure mapping as an object of the super-categorical type since it assigns a possibly proper class of structures with each vocabulary,
- the semantics as a bifunctor from the categories of structures and formulas to the category of truth values,
- DLTP itself as an object of the super-categorical type since it contains the structure mapping.

Syntax, structures, truth values and semantics are defined in the following sections.

3.3 The Standard Vocabulary

The standard vocabulary contains all symbols that are needed to express and describe Promela programs. While it is promising to extend the vocabulary, it is not sensible to omit symbols: The standard vocabulary can be seen as the minimal vocabulary that is necessary to express the syntax and semantics of Promela. In the comments the intended interpretation of the symbols is given.

Definition 5. *The* standard vocabulary *is given by the following definitions.*

Sorts The elementary and the logical sorts are given by: $\Sigma_E = \{I, C\}$ and $\Sigma_L = \{Q, P\}$

Objects of the sorts I, C, Q and P are called integers, channels, queues and processes, respectively.

Function symbols The elements of the sets FS_{RP}, FS_{NP}, FS_{RL} and FS_{NL} and their signatures are as follows.

The rigid program function symbols and their signatures are:

- c_n, $\mathrm{Sign}(c_n) = I$ for $n \in \mathbb{Z}$; *glob* is an alias for c_{-1} $\boxed{1}$
- $+, -, \cdot, /$, $\mathrm{Sign}(+) = \mathrm{Sign}(-) = \mathrm{Sign}(\cdot) = \mathrm{Sign}(/) = I\ I\ I$ $\boxed{2}$
- $elem_i^S$, $\mathrm{Sign}(elem_i^S) = \ S\ S_i$ for $S \in \Sigma_P$ and $i \in [1; \mathrm{Length}(S)]$ $\boxed{3}$
- $tuple^S$, $\mathrm{Sign}(tuple^S) = S_1\ \dots\ S_n\ S$ for $S \in \Sigma_P$ $\boxed{4}$
- ini^S, $\mathrm{Sign}(ini^S) = S$ for $S \in \Sigma_P$ $\boxed{5}$

The non-rigid program function symbols and their signatures are:

- var^S, $\mathrm{Sign}(var^S) = I\ I\ S$ for $S \in \Sigma_P$ $\boxed{6}$
- $nextpid$, $\mathrm{Sign}(nextpid) = I$ $\boxed{7}$
- $lengthc$, $\mathrm{Sign}(lengthc) = C\ I$ $\boxed{8}$
- $pollfirst^{\sigma,S}, pollany^{\sigma,S}$, $\mathrm{Sign}(pollfirst^{\sigma,S}) = \mathrm{Sign}(pollany^{\sigma,S}) = C\ S_{i_1}\ \dots\ S_{i_r}\ I$ for $S \in \Sigma_P$ and[3] $\sigma = \{i_1, \dots, i_r\} \subseteq [1; \mathrm{Length}(S)]$ $\boxed{9}$.

The rigid logical function symbols and their signatures are:

- $\perp^S$, $\mathrm{Sign}(\perp^S) = S$ for $S \in \Sigma$ $\boxed{10}$
- nil^C, $\mathrm{Sign}(nil^C) = C$ $\boxed{11}$
- $incchan$, $\mathrm{Sign}(incchan) = C\ C$ $\boxed{12}$

[3]Whenever the notation $\{i_1, \dots, i_r\}$ is used to define a set the elements $i_1, \dots, i_r$ are assumed to be pairwise different.

- $()^{m,S}$, $\mathrm{Sign}(()^{m,S}) = Q$ for $m \in \mathbb{N}$, $S \in \Sigma_P$ $\boxed{13}$
- cap, $\mathrm{Sign}(cap) = Q\ I$ $\boxed{14}$
- $length$, $\mathrm{Sign}(length) = Q\ I$ $\boxed{15}$
- $remove$, $\mathrm{Sign}(remove) = Q\ I\ Q$ $\boxed{16}$
- $insert^S$, $\mathrm{Sign}(insert^S) = Q\ S\ I\ Q$ for $S \in \Sigma_P$ $\boxed{17}$
- $read^S$, $\mathrm{Sign}(read^S) = Q\ I\ S$ for $S \in \Sigma_P$ $\boxed{18}$
- $sendpos^S$, $\mathrm{Sign}(sendpos^S) = Q\ S\ I$ for $S \in \Sigma_P$ $\boxed{19}$
- $recpos^{\sigma,S}$, $\mathrm{Sign}(recpos^{\sigma,S}) = Q\ S_{i_1}\ \ldots\ S_{i_r}\ I$ for $S \in \Sigma_P$ and $\sigma = \{i_1, \ldots, i_r\} \subseteq [1; \mathrm{Length}(S)]$ $\boxed{20}$
- run, fin, $\mathrm{Sign}(run) = \mathrm{Sign}(fin) = P$ $\boxed{21}$

The non-rigid logical function symbols and their signatures are:

- $proc$, $\mathrm{Sign}(proc) = I\ P$ $\boxed{22}$
- $cont$, $\mathrm{Sign}(cont) = C\ Q$ $\boxed{23}$
- $nextchan$, $\mathrm{Sign}(nextchan) = C$ $\boxed{24}$

The primary non-rigid function symbols are: $FS_{prim} = \{var^S | S \in \Sigma_P\} \cup \{cont, nextchan, proc, nextpid\}$

$\boxed{1}$ Constant symbols for all integers; *glob* is used as the process ID of an artificial process that declares the global variables.

$\boxed{2}$ the usual arithmetical operators; logical and comparison operators can be defined from them

$\boxed{3}$ selects an element from a tuple

$\boxed{4}$ composes a tuple from terms of elementary data types

$\boxed{5}$ initial values for all program sorts

$\boxed{6}$ $var^S(p, i)$ gives the i-th program variable of the sort S of the p-th process.

$\boxed{7}$ gives the next free process ID

$\boxed{8}$ gives the length of the queue that is indexed by a channel

$\boxed{9}$ allow to check without side effects whether a receive operation is enabled

$\boxed{10}$ Undefined values for all sorts; functions with undefined arguments return undefined results, and predicates with undefined arguments are false except for the equality and the comparison of two undefined arguments.

$\boxed{11}$ Channels are pointers to queues. nil^C is the null pointer.

$\boxed{12}$ increments a channel

$\boxed{13}$ the empty queue of capacity m and sort S

$\boxed{14}$ gives the capacity of a queue

$\boxed{15}$ gives the length of a queue

$\boxed{16}$ removes an element at a certain position of a queue

$\boxed{17}$ inserts an element into the queue at a certain position

$\boxed{18}$ reads an element at a certain position from a queue

$\boxed{19}$ gives the position at which an element is inserted for sorted sending

$\boxed{20}$ gives the position from which an element is read for random receive

$\boxed{21}$ two process states for running and finished processes

$\boxed{22}$ gives the execution state of a process with a given ID

$\boxed{23}$ dereferences a channel giving the queue it points to

$\boxed{24}$ gives the next free channel

Predicate Symbols The predicate symbols and their signatures are:

- $=^S$, $\mathrm{Sign}(=^S) = S\ S$ for $S \in \Sigma$ $\boxed{1}$
- $\leq^S$, $\mathrm{Sign}(\leq^S) = S\ S$ for $S \in \Sigma_P$ $\boxed{2}$
- sor^S, $\mathrm{Sign}(sor^S) = Q$ for $S \in \Sigma_P$ $\boxed{3}$
- $match^{\sigma,S}$, $\mathrm{Sign}(match^{\sigma,S}) = S\ S_{i_1}\ \ldots\ S_{i_r}$
  where $S \in \Sigma_P$ and $\sigma = \{i_1, \ldots, i_r\} \subseteq [1; \mathrm{Length}(S)]$ $\boxed{4}$

$\boxed{1}$ equality for all sorts without special treatment of undefined arguments

$\boxed{2}$ Comparison for all program sorts; this is an ordering that is total except for the symbol $\perp^S$ which is only comparable to itself.

$\boxed{3}$ tests whether a queue has the sort S

$\boxed{4}$ Pattern matching: Those components of the first argument that are indexed by σ are matched against the pattern which is given by the other arguments.

---

4 Syntax

In this section terms of the sort S for $S \in \Sigma$, programs and formulas are defined. All these definitions are relative to a fixed vocabulary

$$((\Sigma_E, \Sigma_L), (FS_{RP}, FS_{NP}, FS_{RL}, FS_{NL}), FS_{prim}, PS, \mathrm{Sign})$$

and Σ, Σ_P and FS are defined as in the definition of vocabularies. Then an example is given and substitutions are defined.

Interdependence of the Definitions The definitions of terms, programs and formulas depend on each other in the following way:

- The sets of terms contain conditional terms of the form $[C?t : t']$ where C is a formula, the guard. In particular conditional terms may be nested.
- But formulas that occur in terms may not contain programs.
- Naturally programs contain terms.
- But the conditional terms that occur in programs may only contain simple equality formulas.
- Formulas contain programs via the modal operators and of course terms.

All objects are well-defined by a parallel inductive definition. For readability the definition is split into separate but interdependent definitions of terms, programs and formulas.

Alphabet and Occurrences Terms, programs and formulas are defined as words over the alphabet[4] that contains the following symbols:[5]

- the elements of FS, the logical variables as defined below and the symbols , ( ) [ ] ? and : (for terms)
- the symbols ; | : , and the natural numbers (for tags and the structure of programs)
- the symbol := (for assignments)
- the symbols ! and !! (for send commands)
- the symbols ? ?? ? < and ?? < (for receive commands)

[4]For programs a structural definition that allows to directly access the components of a program is given, too.
[5]Some symbols are listed twice because they are used with different meanings.

- the symbols ELSE and BREAK
- the symbol RUN and the elements of a (countable) set **P** as defined below (for run commands)
- the symbols END ( ) and , (for end commands)
- the symbols IF, FI, DO, OD and :: (for if and do commands)
- the symbols ATOMIC { and } (for atomic commands)
- the symbols BT and EC (for labels of commands)
- the elements of PS, the logical variables as defined below and the symbols ( ) , $\neg$ $\wedge$ $\vee$ $\forall$ $\exists$ [ ] $\langle$ and $\rangle$ (for formulas)

Furthermore it is assumed that the alphabet also contains symbols to express the following set theoretical objects: finite sets of natural numbers and tuples of terms over such sets (for receive commands) and partial mappings with a finite domain from program sorts and natural numbers to terms[6] (for run commands).

Then terms, programs and formulas are finite sequences over this alphabet, called words. The elements of a word are written without separation symbols, and the composition of words is denoted by juxtaposition. The empty word is denoted by ϵ.

The word v occurs in the word w or w contains the word v if there are words u and u' such that $w = uvu'$. Then v is said to be an occurrence within w. Replacing this occurrence with v' gives the word $uv'u'$.

4.1 Terms

Logical Variables

Definition 6. *The* logical variables *of the sort S are the elements of the set $LV(S) = \{x_i^S | i \in \mathbb{N}^*\}$ for $S \in \Sigma$. The set $\bigcup\limits_{S\in\Sigma} LV(S)$ is denoted by LV.*

Terms

Definition 7. *The sets $Terms(S)$ of* terms *of the sort S are built up inductively from logical variables and function symbols for all sorts $S \in \Sigma$:*

1. *$LV(S) \subseteq Terms(S)$*
2. *If $f \in FS$, $\mathrm{Sign}(f) = S^1 \ \ldots \ S^n \ S$ for $n \in \mathbb{N}$ and $t_i \in Terms(S_i)$ for $i \in [1;n]$, then*

   $$f(t_1, \ldots, t_n) \in Terms(S).$$
3. *If C is a first-order formula and if $t, t' \in Terms(S)$, then $[C?t : t'] \in Terms(S)$. Such terms are called conditional terms, C is called the guard of the conditional term.*
4. *$(Terms(S))_{S\in\Sigma}$ is the smallest family of sets that is closed under the above rules.*[7]

Abbreviations

Definition 8. *The sets $FTerms(S)$ of* free terms *of the sort $S \in \Sigma$ are defined as $Terms(S)$, but without clause* 1. $\boxed{1}$

Simple terms *are defined as terms, but without the clause* 3. $\boxed{2}$

Rigid terms *are defined as terms, but with $FS_{RP} \cup FS_{RL}$ instead of FS in clause* 2. $\boxed{3}$

The sets $PTerms(S)$ of program terms *of the sort $S \in \Sigma_P$ are defined as $Terms(S)$, but without the clause* 1, *with $FS_{RP} \cup FS_{NP}$ instead of FS in clause* 2 *and with the restriction that C is of the form $\neg s =^{S'} ini^{S'}$ for $S' \in \Sigma_P$ and $s \in PTerms(S')$ in clause* 3. $\boxed{4}$

For $S \in \Sigma_P$ the set $\{var^S(t, t') | t, t' \in PTerms(I)\}$ is denoted by $PV(S)$. These terms are called program variables *of the sort S.*

[6]Since a term uniquely determines its sort such a mapping can be denoted as a finite relation between natural numbers and terms.

[7]"smallest" means the componentwise intersection of all families that satisfy the rules.

For the standard vocabulary two further abbreviations are used: For $\diamond \in \{+, -, \cdot, /\}$ infix notation is used, i. e. for $t, t' \in Terms(I)$ the word $\diamond(t, t')$ is written $(t \diamond t')$ and outermost brackets of a term are omitted. And for addition and subtraction associative brackets are omitted, i. e. for $t_1, t_2, t_3 \in Terms(I)$ and $\diamond, \circ \in \{+, -\}$ the term $((t_1 \diamond t_2) \circ t_3)$ is abbreviated by $(t_1 \diamond t_2 \circ t_3)$

$\boxed{1}$ Free terms do not contain logical variables. Their interpretation does not depend on the assignment.

$\boxed{2}$ Simple terms are the terms as in classical many-sorted first-order predicate logic.

$\boxed{3}$ Rigid terms have the same interpretation in all states.

$\boxed{4}$ Program terms are free, do not contain formulas except for comparison to zero and contain only program function symbols. Only program terms may occur in programs.

4.2 Programs

In this section programs are first defined structurally as high-level objects. Then a grammar is given that defines programs as words over the used alphabet. The translation between the two definitions is not given. Instead the structural definition is used in this work and the grammar definition is considered to give the denotations of the programs. The definitions of process types and syntactical abbreviations are given in their own sections.

4.2.1 Structural Definition

Commands and Command Sequences

Definition 9. *Commands and command sequences are defined simultaneously as follows:*

- *A command is an element of $\underline{\text{command}}$ which is the disjunct union of the following sets: $\underline{\text{assignment}}$, $\underline{\text{expression}}$, $\underline{\text{send}}$, $\underline{\text{receive}}$, $\underline{\text{break}}$, $\underline{\text{else}}$, $\underline{\text{run}}$, $\underline{\text{end}}$, $\underline{\text{if}}$, $\underline{\text{do}}$ and $\underline{\text{atomic}}$. The respective set is called the type of the commands it contains. The union of the types $\underline{\text{assignment}}$, $\underline{\text{expression}}$, $\underline{\text{send}}$, $\underline{\text{receive}}$, $\underline{\text{break}}$, $\underline{\text{else}}$, $\underline{\text{run}}$ and $\underline{\text{end}}$ is called $\underline{\text{elementary}}$. Elementary commands are not composed of other commands.*
- *A command sequence is a finite non-empty sequence of commands that does not contain a command of the type $\underline{\text{end}}$.*
- *A command of the type $\underline{\text{assignment}}$ is a pair of a program variable $X \in PV(S)$ and a program term $t \in PTerms(S)$ for $S \in \Sigma_P$.*
- *A command of the type $\underline{\text{expression}}$ is a program term $t \in PTerms(S)$ for $S \in \Sigma_P$.*
- *A command c of the type $\underline{\text{send}}$ for the sort $S = Sor(c) \in \Sigma_P$ is a tuple of*
  - *a program variable $Chan(c) \in PV(C)$,*
  - *a type $Type(c) \in \{!, !!\}$,*
  - *a program term $Arg(c) \in PTerms(S)$.*
- *A command c of the type $\underline{\text{receive}}$ for the sort $S = Sor(c)$, where $S = S_1 \times \ldots \times S_n \in \Sigma_P$, is a tuple of*
  - *a program variable $Chan(c) \in PV(C)$,*
  - *a type $Type(c) \in \{?, ??, ?<, ??<\}$,*
  - *a set $Pos(c) = \sigma \subseteq [1; n]$,*
  - *a tuple $Var(c) = (X_i)_{i \in \sigma'}$ such that $X_i \in PV(S_i)$ for $i \in \sigma' = [1; n] \setminus \sigma$,*
  - *a tuple $Arg(c) = (t_i)_{i \in \sigma}$ such that $t_i \in PTerms(S_i)$ for $i \in \sigma$.*
- *The types $\underline{\text{break}}$ and $\underline{\text{else}}$ contain only one element each.*
- ***P** is a countable set of process types, this set and the mapping $NmbPar$ are defined below. Then a command c of the type $\underline{\text{run}}$ is a tuple of*
  - *a process type $Proc(c) \in \mathbf{P}$,*
  - *a mapping $Par(c)$, which maps pairs (S, i) such that $S \in \Sigma_P$ and $i \in [1; NmbPar(Proc(c), S)]$ to program terms in $PTerms(S)$.*

- *A command of the type* <u>end</u> *is a pair of a process type* $\mathrm{PT} \in \mathbf{P}$ *and a program term of the sort* I.[8]
- *A command* c *of the type* <u>if</u> *or* <u>do</u> *is a finite non-empty sequence of command sequences, called options. Only the last option may start with a command of the type* <u>else</u>*; if so it is called the else option. The number of non-else options of* c *is denoted by* $opt(c) \in \mathbb{N}$*, the number of else options by* $optelse(c) \in \{0,1\}$*. The options are denoted by* $c[1], \ldots, c[opt(c) + optelse(c)]$ *and, if an else option exists,* $c[else]$ *abbreviates* $c[opt(c) + optelse(c)]$.

  A command $c \in$ <u>do</u> *may be labelled* BT *(short for break target): This is denoted by* $c = \mathrm{BT}\ c'$.
- *A command* c *of the type* <u>atomic</u> *is a command sequence that is denoted by* $c[body]$*. A command* $c \in$ <u>atomic</u> *may be labelled* EC *(short for exclusive control): This is denoted by* $c = \mathrm{EC}\ c'$.

Examples The command

$$var^C(glob, c_1)\ !\ tuple^S(c_0, c_1)$$

of the sort $S = I \times I$ sends the value $(0,1)$ to the queue that is pointed to by the channel $var^C(glob, c_1)$. The object is inserted at the first position.

The command

$$var^C(c_1, c_1)\ ??\ \{1,3\}\ (var^I(c_1, c_1), var^C(c_1, c_2))\ (c_1, nil^C)$$

of the sort $I \times I \times C \times C$ receives the first element in the queue that is pointed to by the channel $var^C(c_1, c_1)$ that has the value 1 in its first and the value of nil^C in its third component. The second component of this object is assigned to the variable $var^I(c_1, c_1)$ and the fourth component to $var^C(c_1, c_2)$. The object is then removed from the queue.

The command RUN PT p where $p(I,1) = c_2$ and $p(C,1) = nil^C$ instantiates the process type PT and initializes its two parameters of the sorts I and C with 2 and the value of nil^C.

The command $\mathrm{END}(\mathrm{PT}, c_2)$ resets all local variables of the second process.

Programs

Definition 10. *A* process *is defined as a command sequence with the exception that it may contain a command of the type* <u>end</u> *at the last position.*

A program *is pair of a tag* τ *and a finite*[9] *sequence of processes* π*, such that, if* $n = \mathrm{Length}(\pi)$,

$$\tau \in \{\epsilon\} \cup [1;n] \cup \{(i,j) | i,j \in [1,n], i \neq j\}.$$

In only one process an EC *label may occur. The program* (ϵ, π) *is identified with* π.

Prog is the set of programs.

The intuitive meaning of a program with an empty tag (ϵ, π) is that π is a program in the intuitive sense. The intuitive meaning of a tagged program is that the result of the scheduling decision is stored in the program. The program (i, π) means that an asynchronous command from π_i is attempted to be executed next. The program $((i,j), \pi)$ means that the synchronous execution of commands form π_i and π_j is attempted to be executed next.

4.2.2 Grammar

The definition of programs can also be given in the more usual grammar form. The grammar also gives the denotations of programs, command sequences and commands that are used in this work. In the grammar "/" denotes alternatives, "[ ]" denotes optional parts, the start symbol is <u>program</u>, and non-terminal symbols are underlined.

[8] Commands of this type do not exist in Promela. They are used to store the process type and ID during run-time and may only occur at the end of a process. Execution of such a command has priority and resets all program variables of the process.

[9] The empty program is allowed whereas empty processes are forbidden.

program	::=	tag processes
tag	::=	ϵ / N : / N , N :
processes	::=	sequence [; end] | processes / ϵ
sequence	::=	command ; sequence / command
command	::=	assignment / expression / send / receive / break / else / run / if / do / atomic
if	::=	IF options FI
do	::=	[BT] DO options OD
options	::=	:: sequence options / :: sequence
atomic	::=	[EC] ATOMIC { sequence }

assignment	::=	V(S) := T(S)
		 for any $S \in \Sigma_P$

expression	::=	T(S)
		 for any $S \in \Sigma_P$

send	::=	V(C) typesend T(S)
		 for any $S \in \Sigma_P$

receive	::=	V(C) typerec $Pos\ Var\ Arg$ where Pos, Var and Arg are as above
break	::=	BREAK
else	::=	ELSE
run	::=	RUN PT Par where Par is as above
end	::=	END(PT , T(I))
N	::=	n for any $n \in \mathbb{N}^*$
V(S)	::=	X for any $X \in PV(S)$ and any $S \in \Sigma_P$
T(S)	::=	t for any $t \in PTerms(S)$ and any $S \in \Sigma_P$
typesend	::=	! / !!
typerec	::=	? / ?? / ? < / ?? <
PT	::=	p for any $p \in \mathbf{P}$

For simplicity, some conditions are not enforced by the grammar, so that all words that do not satisfy them have to removed from the generated language. These are:

- The natural numbers in the tag must not be larger than the number of processes in the program. And the tag i, j : must satisfy $i \neq j$.
- In only one process an EC label may occur.
- There may only be one option that starts with the command ELSE in an if or do command which must be the last one.

It can be easily seen that there is a natural way to parse a word of this language onto the structural definition of a program given above.

4.2.3 Process Types

Definition 11. *A* process type PT *is a tuple of*

- *a family $(NmbVar(\mathrm{PT}, S))_{S \in \Sigma_P}$ of natural numbers such that $NmbVar(\mathrm{PT}, S) \neq 0$ for only finitely many S, which gives the number of local variables (except for the parameters) for all program sorts,*
- *a mapping $Ini(\mathrm{PT})$ that maps (S, i) where $S \in \Sigma_P$ and $i \in [1; NmbVar(PT, S)]$ to a simple rigid term of any sort, which gives the initializations of the local variables (except for the parameters),*
- *a family $(NmbPar(\mathrm{PT}, S))_{S \in \Sigma_P}$ of natural numbers such that $NmbPar(\mathrm{PT}, S) \neq 0$ for only finitely many S, which gives the number of parameters for all program sorts,*

- *a body* $Body(\mathrm{PT})$ *which is a process with* $\mathrm{Last}(Body(\mathrm{PT})) = \mathrm{END}(\mathrm{PT}, nextpid)$ *or* $\mathrm{Last}(Body(\mathrm{PT})) \notin \underline{\mathrm{end}}$*,*

such that

- *for* $S \in \Sigma_P$ *all program variables* $var^S(t, t')$ *that occur in* $Body(\mathrm{PT})$ *satisfy* $t \in \{glob, nextpid\}$ *and* $t' \in \left\{ c_i | i \in \begin{cases} \mathbb{N}^* & \text{if } t = glob \\ [1; NmbVar(\mathrm{PT}, S) + NmbPar(\mathrm{PT}, S)] & \text{if } t = nextpid \end{cases} \right\}$,
- $decl(\mathrm{PT})$ *is finite where decl is defined by*

$$decl(\mathrm{PT}) = \{\mathrm{PT}\} \bigcup_{Q \in Body(\mathrm{PT})} decl(Q)$$

  where $Q \in Body(\mathrm{PT})$ *means that* Q *is a process type that occurs in* $Body(\mathrm{PT})$*.*

Explanations and intuitive meanings:

- The body does not contain declarations or initializations. It only contains commands.
- Parameters are a special kind of local variables. The remaining local variables and the parameters are indistinguishable in the body. The only difference between the two kinds is the initialization: The parameters are initialized with values that are passed by the calling process; the initial values for all other local variables are given by $Ini(\mathrm{PT})$.
- If $m = NmbVar(\mathrm{PT}, S)$ and $n = NmbPar(\mathrm{PT}, S)$ and if t is the ID of a process of the type PT, the standard structure (see section 8) uses m and n in the following way: The program variables $var^S(t, c_1)$ to $var^S(t, c_m)$ are the normal local variables of the sort S and the program variables $var^S(t, c_{m+1})$ to $var^S(t, c_{m+n})$ are the parameters of the sort S.
- *glob* identifies global and *nextpid* local program variables. When a process type is instantiated the instance receives a process ID that must be the first argument of var^S for $S \in \Sigma$ to access the local program variables. In the definition of the semantics of a program occurrences of *nextpid* are modified (see the definition of $\mathrm{rem}^{\mathrm{A}}$ in section 7.2.1) such that the first argument of var^S is always *glob* or interpreted as this ID.
- The restriction of program variable occurrences excludes most senseless process types. Global program variables must have positive indices, and only those local program variables are allowed that are declared by the process type. However, it is still possible to access global program variables with higher indices than have been declared by the process with ID *glob*. These are undefined if they have not been written by another command before.
- In the standard structure $Ini(\mathrm{PT})(S, i)$ syntactically gives the initial value of the local variable $var^S(t, c_i)$ if t is the ID of a process of the type PT. The phrase "simple rigid term of any sort" in the definition is very surprising. "rigid program term of the sort S" would render the definition more natural and intuitive. And of course if $Ini(\mathrm{PT})(S, i) \notin Terms(S)$ it cannot be directly interpreted as an initial value.

  Therefore the structures must take care that $Ini(\mathrm{PT})$ is used in a sensible way. In the definition of the standard structure for the standard vocabulary the initialization algorithm (page 42) is used.
- $decl(\mathrm{PT})$ gives the set of process types that must be declared if PT is to be used in a program. It must be finite because in an implementation the declaration of the necessary process types must be encoded in some finitary way. Theoretically, if infinite such sets were allowed it would be possible to give a program the execution of which would lead to unboundedly complex (e. g. as to the depth of nesting of composed commands, or the number of components of a composed sort) intermediate programs.

Inspecting the definitions of process types and command sequences shows that both are well-defined and that there are indeed only countably many process types. For convenience, the declaration of the used process types is usually given in some context and then **P** contains only symbols that refer to these declarations.

4.2.4 Abbreviations

Labels The labels EC and BT are used in intermediate programs for the recursive definition of the semantics. The function Label gives the label of a command:

$$\text{Label}(c) = \begin{cases} \text{L} & \text{if } c = \text{L}\ c';\ L \in \{\text{BT}, \text{EC}\},\ c' \in \underline{\text{command}} \\ \epsilon & \text{otherwise} \end{cases}$$

The function Dropec removes a leading EC label from all processes of a program:

$$\text{Dropec}(\pi) = \begin{cases} c; \text{Rest}(\pi_1) & \text{if Length}(\pi) = 1,\ \text{First}(\pi_1) = \text{EC}\ c,\ c \in \underline{\text{command}} \\ \pi & \text{if Length}(\pi) = 1,\ \text{Label}(\text{First}(\pi_1)) \neq \text{EC} \\ \text{Dropec}(\pi_1) \mid \text{Dropec}(\text{Rest}(\pi)) & \text{otherwise} \end{cases}$$

Similarly the function Dropbt removes a leading BT label from a process:

$$\text{Dropbt}(s) = \begin{cases} c; \text{Rest}(s) & \text{if First}(s) = \text{BT}\ c,\ c \in \underline{\text{command}} \\ s & \text{otherwise} \end{cases}$$

Replacement of Processes For a program π, $i \in [1; \text{Length}(\pi)]$ and a finite sequence of commands s, $\pi^i(s)$ denotes the program where the i-th element of π is replaced with s if s is non-empty and removed if s is empty:

$$\pi^i(s) = \begin{cases} \text{Insert}(\text{Remove}(\pi, \{i\}), s, i) & \text{if } s \neq () \\ \text{Remove}(\pi, \{i\}) & \text{if } s = () \end{cases}$$

Atomic Commands The mapping ECAtomic maps a finite non-empty sequence of commands to a command of the type <u>atomic</u> with EC label, but is also defined for an empty sequence of commands:

$$\text{ECAtomic}(s) = \begin{cases} \text{EC ATOMIC } \{s\} & \text{if } s \neq () \\ () & \text{otherwise} \end{cases}$$

Program Variables The notation of program variables is complicated. In order to provide a more intuitive notation the following convention is used: If a program variable $var^S(t, c_i)$ for $S \in \Sigma_P$, $t \in PTerms(I)$ and $i \in \mathbb{N}^*$ occurs in a process s with $\text{Last}(s) = \text{END}(\text{PT}, t)$ for some $\text{PT} \in \mathbf{P}$, this program variable is abbreviated by X_i^S.

This allows to use the same symbol for the i-th local variable of the sort S in all processes. The abbreviation depends on the context: The correct process ID must be stored in an <u>end</u> command at the end of the process.

The corresponding abbreviation for global variables is much simpler: The program variable $var^S(glob, c_i)$ for $S \in \Sigma_P$ and $i \in \mathbb{N}^*$ is abbreviated by Y_i^S.

For example the program

$$var^I(c_3, c_1);\ var^C(glob, c_2);\ \text{END}(\text{PT}, c_3) \mid$$

$$var^I(c_5, c_1);\ var^C(glob, c_2);\ \text{END}(\text{PT}, c_5)$$

is abbreviated by

$$X_1^I;\ Y_2^C;\ \text{END}(\text{PT}, c_3) \mid$$

$$X_1^I;\ Y_2^C;\ \text{END}(\text{PT}, c_5).$$

4.3 Formulas

Atomic formulas

Definition 12. Atomic formulas *are the logical constants true and false and all instantiations of predicate symbols with terms of sorts according to the signature of the predicate symbol:*

$$AtForm = \{true, false\} \cup \{P(t_1, \dots, t_n) | P \in PS, \text{Sign}(P) = S^1\ \dots\ S^n, t_i \in Terms(S_i) \text{ for } i \in [1; n]\}.$$

Updates

Definition 13. *An* update *is a triple* $(f, (t_1, \ldots, t_n), t)$ *where* $f \in FS_{prim}$ *with* $\mathrm{Sign}(f) = S^1 \; \ldots \; S^n \; S$ *for* $n \in \mathbb{N}$, $t_i \in Terms(S_i)$ *for* $i \in [1; n]$, *and* $t \in Terms(S)$.

Modal Operators

Definition 14. *Oper denotes the set* $\{[\cdot], \langle\cdot\rangle, [[\cdot]], [\langle\cdot\rangle], \langle[\cdot]\rangle, \langle\langle\cdot\rangle\rangle\}$. *The elements of Oper map a program to a unary operator on the set of formulas.*

The operators $[]$ and $\langle\rangle$ correspond to the operators $\Box$ and $\Diamond$ of modal logic. The formulas $[[\cdot]]F$, $[\langle\cdot\rangle]F$, $\langle[\cdot]\rangle F$ and $\langle\langle\cdot\rangle\rangle F$ correspond to the CTL formulas (see for example [9]) *all* F *until* $false$, *all true until* F, *ex* F *until* $false$ and *ex true until* F, respectively.

Formulas

Definition 15. *The set Form of* formulas *is built up inductively from atomic formulas using the unary operators* $\neg$, $\forall x$, $\exists x$ *(quantifiers for logical variables* x*),* $M(\pi)$ *(modal operators for* $M \in Oper$ *and programs* π*) and* u *(for updates* u*) and the binary operators* $\wedge$ *and* $\vee$*:*

- $AtForm \subseteq Form$
- *If* $F, G \in Form$, *then* $\neg F, (F \wedge G), (F \vee G) \in Form$.
- *If* $F \in Form$, $x \in LV$, *then* $\forall x F$, $\exists x F \in Form$.
- *If* $M \in Oper$, $\pi \in Prog$ *and* $F \in Form$, *then* $M(\pi)F \in Form$.
- *If* $F \in Form$ *and* u *is an update, then* $u\ F \in Form$.
- *Form is the smallest set that is closed under the above rules.*

Free Variables and Closed Formulas

Definition 16. *FV maps terms, updates and formulas to the set of their* free variables. *It is defined by*

- $FV(x) = \{x\}$ *if* $x \in LV$
- $FV(f(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} FV(t_i)$ *if* $f \in FS$
- $FV([C?t_1 : t_2]) = FV(C) \cup FV(t_1) \cup FV(t_2)$
- $FV(true) = FV(false) = \emptyset$
- $FV(P(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} FV(t_i)$ *if* $P \in PS$
- $FV(\neg F) = FV(F)$
- $FV(F \diamond G) = FV(F) \cup FV(G)$ *if* $\diamond \in \{\wedge, \vee\}$
- $FV(Qx\ F) = FV(F) \setminus \{x\}$ *if* $Q \in \{\forall, \exists\}$ *and* $x \in LV$
- $FV(M(\pi)F) = FV(F)$ *if* $M \in Oper$ *and* $\pi \in Prog$
- $FV(u) = FV(t_1) \cup \ldots \cup FV(t_n) \cup FV(t)$ *for an update* $u = (f, (t_1, \ldots, t_n), t)$
- $FV(U) = \bigcup_{i=1}^{n} FV(U_i)$ *for a sequence of updates* U *with length* n
- $FV(u\ F) = FV(u) \cup FV(F)$ *for an update* u

A logical variable that occurs in a formula F *immediately behind a quantifier is called* bound *in* F.

A formula F *is called* closed *if* $FV(F) = \emptyset$.

Abbreviations The following abbreviations are intuitively clear and are only defined for completeness.

Definition 17. *For $n \in \mathbb{N}$, $n \geq 3$ and $\diamond = \wedge$ or $\diamond = \vee$, the word $(\dots(F_1 \diamond F_2)\dots \diamond F_n)$ is abbreviated by $(F_1 \diamond F_2 \dots \diamond F_n)$ and by $\bigwedge_{i=1}^{n} F_i$ or $\bigvee_{i=1}^{n} F_i$, respectively.*

$F_1 \wedge \dots \wedge F_0$ is an abbreviation of $true$, and $G_1 \vee \dots \vee G_0$ is an abbreviation of $false$.

The word $(\neg F \vee G)$ is abbreviated by $(F \rightarrow G)$.

The formula (F) is abbreviated by F.

If $U = (u_1, \dots, u_n)$ is a finite sequence of updates, then the formula $u_1 \ \dots \ u_n \ F$ is abbreviated by $U \ F$.

The universal quantifier with a colon instead of a logical variable gives the universal closure of a formula: $\forall : \ F$ abbreviates $\forall x_1 \dots \forall x_n F$ if $FV(F) = \{x_1, \dots, x_n\} \subseteq LV$.

For the standard vocabulary two further abbreviations are used: For $S \in \Sigma$ and $P = =^S$ or $S \in \Sigma_P$ and $P = \leq^S$ infix notation is used, i. e. for $t, t' \in Terms(S)$ the word $P(t, t')$ is written tPt'.

Restricted Sets of Formulas

Definition 18. *The following properties of formulas are defined:*

- *A formula is called* rigid *if it does not contain non-rigid function symbols or programs.*
- *A formula or is called* first-order *if it does not contain modal operators or updates. (Conditional terms are allowed.)*

4.4 Example

The following example program is a slightly modified version of the example given in [16, pg. 98]. In Promela syntax it reads

```
int x=41;

proctype A(chan q1)
{
        chan q2;
        q1?q2;
        q2!42
}

proctype B(chan qforb)
{
        qforb?x;
}

init
{
        chan qname = [1] of { chan };
        chan qforb = [1] of { int };
        run A(qname);
        run B(qforb);
        qname!qforb
}
```

Executing this Promela program will do the following: The global variable x is initialized with the value 41 and the init process type is instantiated. Two channels *qname* and $qforb$ are declared and initialized with a queue of the capacity 1 and sort *chan* or *int* respectively. The init process instantiates a process A with parameter *qname* which initializes $q1$; A declares the channel $q2$. The first command of A is not executable since $q1$ is empty. Therefore the init process is scheduled and instantiates the process type B with parameter $qforb$ which initializes the local variable $qforb$ of B. The first command of B is not executable either and again the init process is scheduled: It sends the channel $qforb$ that has been passed to B to the channel *qname* that has been passed to A. Now the first command of A becomes executable, A receives this channel and sends the integer 42 to it. Now the command of B becomes executable, B receives the integer sent by A and assigns it to x.

For the translation into the syntax of DLTP the standard vocabulary is used. All declarations and initializations of variables and parameters must be captured by the definitions of $NmbVar$, Ini and $NmbPar$ for the used process types. A process type GLOBAL is added, which declares the global variables and instantiates the init process. Also the variable and sort symbols must be changed.

Since all local variables are of the sort C, the superscript C of local variables is left out. Then the process types $\text{GLOBAL}, \text{init}, \text{A}, \text{B} \in \mathbf{P}$ are given by

- GLOBAL:
  - $NmbVar(\text{GLOBAL}, S) = \begin{cases} 1 & \text{if } S = I \\ 0 & \text{otherwise} \end{cases}$
    for $S \in \Sigma_P$ $\boxed{1}$
  - $Ini(\text{GLOBAL})(I, 1) = c_{41}$ $\boxed{2}$
  - $NmbPar(\text{GLOBAL}, S) = 0$ for $S \in \Sigma_P$
  - $Body(\text{GLOBAL}) = \text{RUN init}$
- init:
  - $NmbVar(\text{init}, S) = \begin{cases} 2 & \text{if } S = C \\ 0 & \text{otherwise} \end{cases}$
    for $S \in \Sigma_P$ $\boxed{3}$
  - $Ini(\text{init})(C, 1) = ()^{1,C}$,
    $Ini(\text{init})(C, 2) = ()^{1,I}$ $\boxed{4}$
  - $NmbPar(\text{init}, S) = 0$ for $S \in \Sigma_P$
  - $Body(\text{init}) =$

    RUN A p_1;
    RUN B p_2;
    $X_1 \,!\, X_2$;
    END(init, $nextpid$)

    where $p_1(C, 1) = X_1$ and $p_2(C, 1) = X_2$.
- A:
  - $NmbVar(\text{A}, S) = \begin{cases} 1 & \text{if } S = C \\ 0 & \text{otherwise} \end{cases}$
    for $S \in \Sigma_P$
  - $Ini(\text{A})(C, 1) = \perp^C$ $\boxed{5}$
  - $NmbPar(\text{A}, S) = \begin{cases} 1 & \text{if } S = C \\ 0 & \text{otherwise} \end{cases}$
    for $S \in \Sigma_P$ $\boxed{6}$
  - $Body(\text{A}) =$

    $\overbrace{X_2 \,?\, \emptyset\ (X_1)\ ()}^{\boxed{7}}$;
    $X_1 \,!\, c_{42}$;
    END(A, $nextpid$)
- B:
  - $NmbVar(\text{B}, S) = 0$ for $S \in \Sigma_P$
  - The domain of $Ini(\text{B})$ is empty.
  - $NmbPar(\text{B}, S) = \begin{cases} 1 & \text{if } S = C \\ 0 & \text{otherwise} \end{cases}$
    for $S \in \Sigma_P$

– *Body*(B) =

$X_1 \;?\; \emptyset \;(Y_1^I)\;();\; \boxed{8}$
$\text{END}(\text{B}, \mathit{nextpid})$

The restrictions of variable occurrences are met for the four process types. Also for example
$\mathit{decl}(\text{GLOBAL}) = \{\text{GLOBAL}, \text{init}, \text{A}, \text{B}\}$ and
$\mathit{decl}(\text{init}) = \{\text{init}, \text{A}, \text{B}\}$

$\boxed{1}$ One global program variable of the sort I ...

$\boxed{2}$ ... is initialized with c_{41}.

$\boxed{3}$ Two local program variables of the sort C are declared.

$\boxed{4}$ Here terms of the sort Q are used to initialize the channels. These queues are the queues the new channels initially point to. See the interpretation of the program in section 8.3.1 for the details.

$\boxed{5}$ One local program variable that is not initialized. This variable will be assigned the value of ini^C when the process is instantiated.

$\boxed{6}$ One parameter of the sort C.

$\boxed{7}$ The notation of commands $c \in \underline{\text{receive}}$ may be confusing: $\mathit{Pos}(c) = \emptyset$ states that there is no pattern matching. And $\mathit{Arg}(c) = ()$ is the empty pattern. All this command does is to read an element from the channel X_2, which is a parameter and has been defined by the instantiating process, to the local program variable X_1, for which no special initial value has been provided.

$\boxed{8}$ This command changes the value of Y_1^I from 41 to 42.

Then the DLTP program that corresponds to the Promela program above is RUN GLOBAL.[10]

Examples for formulas are:

- $[\text{RUN GLOBAL}]\; Y_1^I =^I c_{42}$
- $\langle\langle \text{RUN GLOBAL} \rangle\rangle\; \exists x_1^C\; \mathit{read}^I(\mathit{cont}(x_1^C), c_1) =^I c_{42}$

4.5 Substitutions

4.5.1 Logical Substitutions

Definition 19. *A* logical substitution *σ is a set of pairs (x,t) where $x \in LV(S)$ and $t \in \mathit{Terms}(S)$ for some $S \in \Sigma$ such that there are no pairs (x,t) and (x,t') in σ satisfying $t \neq t'$. The logical substitution $\{(x_1,t_1),\ldots,(x_n,t_n)\}$ is denoted by $\{t_1/x_1,\ldots,t_n/x_n\}$.*

A logical substitution σ induces a mapping form Form to Form and from Terms(S) to Terms(S) for $S \in \Sigma$ in the following way. The image $\sigma(x)$ of an object x under this mapping is abbreviated by σx.

For terms s:

$$\sigma s = \begin{cases} t & \text{if } s = x,\ (x,t) \in \sigma \\ f(\sigma t_1,\ldots,\sigma t_n) & \text{if } s = f(t_1,\ldots,t_n),\ f \in FS \\ [\sigma C?\sigma t_1 : \sigma t_2] & \text{if } s = [C?t_1 : t_2] \\ s & \text{otherwise} \end{cases}$$

For formulas F:

$$\sigma F = \begin{cases} F & \text{if } F \in \{\mathit{true}, \mathit{false}\} \\ P(\sigma t_1,\ldots,\sigma t_n) & \text{if } F = P(t_1,\ldots,t_n),\ P \in PS \\ \neg\sigma G & \text{if } F = \neg G \\ \sigma G \diamond \sigma H & \text{if } F = G \diamond H,\ \diamond \in \{\wedge,\vee\} \\ Qx\ \sigma' G & \text{if } F = Qx\ G,\ Q \in \{\forall,\exists\},\ \sigma' = \{(y,t) \in \sigma | y \neq x\}\ \boxed{1} \\ M(\pi)\sigma G & \text{if } F = M(\pi)G,\ M \in \mathit{Oper}\ \boxed{2} \\ (f,(\sigma r_1\ldots,\sigma r_n),\sigma r)\ \sigma G & \text{if } F = (f,(r_1,\ldots,r_n),r)\ G \end{cases}$$

[10]It must be executed form the origin of the standard structure (or at least from a state in which *nextpid* has the value −1 and *nextchan* is defined).

4.5.2 Update Substitutions

Update substitutions affect non-rigid function symbols in a similar way as logical substitutions affect logical variables. There is an important difference for an update $u = (f, (t_1, \ldots, t_n), t)$ in the case $\mathrm{Sign}(f) = S^1 \ \ldots \ S^n \ S$ and $n \geq 1$: The change only applies to the value of $f(t_1, \ldots, t_n)$ and not to the whole interpretation of the function symbol f.

In this case the aliasing problem means that it is syntactically not easily decidable whether or not $f(s_1, \ldots, s_n)$ is affected by u, which is the case if t_i and s_i have the same semantics for $i \in [1; n]$. This problem is solved by introducing conditional terms that guarantee that after applying the update substitution the intended semantics is achieved.

Definition 20. *An update $u = (f, (t_1, \ldots, t_n), t)$ with $\mathrm{Sign}(f) = S^1 \ \ldots \ S^n \ S$ induces a mapping σ_u between both terms and first-order formulas, called an* update substitution*, by the following definition.*

σ_u is defined for terms r by

$$\sigma_u(r) = \begin{cases} r & \text{if } r = x \in LV \\ t & \text{if } r = c \in FS,\ c = f \\ r & \text{if } r = c \in FS,\ c \neq f \\ [\sigma_u(r_1) =^{S^1} t_1 \wedge \ldots \wedge \sigma_u(r_n) =^{S^n} t_n \ ?\ t \ :\ f(\sigma_u(r_1), \ldots, \sigma_u(r_n))] & \text{if } r = f(r_1, \ldots, r_n)\ \boxed{3} \\ g(\sigma_u(r_1), \ldots, \sigma_u(r_m)) & \text{if } r = g(r_1, \ldots, r_m),\ g \neq f \\ [\sigma_u(C) ? \sigma_u(r_1) : \sigma_u(r_2)] & \text{if } r = [C ? r_1 : r_2] \end{cases}$$

σ_u is defined for first-order formulas F after applying the following variable renaming algorithm to F:

```
For all sorts $S \in \Sigma$ (in arbitrary order)
    Let $x \in FV(u)$ be the logical variable of the sort $S$ with the smallest index that occurs bound in $F$.
    Continue with the next $S$ if no such $x$ exists.
    Let $z$ be any logical variable of the sort $S$ that does not occur in $F$. Set $F$ to $\{x/z\}F$.
    Let $y$ be the logical variable of the sort $S$ with the smallest index that does not occur in $F$ or $u$.
    Set $F$ to the formula that arises by replacing all occurrences of $x$ in $F$ with $y$.
    Set $F$ to $\{z/x\}F$.
Next $S$
```

Then it can be assumed that no logical variable that occurs bound in F is a free variable of u and

$$\sigma_u(F) = \begin{cases} F & \text{if } F \in \{true, false\} \\ P(\sigma_u(r_1), \ldots, \sigma_u(r_m)) & \text{if } F = P(r_1, \ldots, r_m),\ P \in PS \\ \neg\sigma_u(G) & \text{if } F = \neg G \\ (\sigma_u(G) \diamond \sigma_u(H)) & \text{if } F = (F \diamond G),\ \diamond \in \{\wedge, \vee\} \\ Qx\ \sigma_u(G) & \text{if } F = Qx\ G,\ Q \in \{\forall, \exists\},\ x \in LV\ \boxed{4} \end{cases}$$

A finite sequence U of updates of the length n induces an analogous mapping σ_U by

$$\sigma_U = \sigma_{U_1} \circ \ldots \circ \sigma_{U_n}$$

$\boxed{1}$ Logical substitutions only affect the free variables of a formula.

$\boxed{2}$ Programs do not contain logical variables and are not affected by logical substitutions.

$\boxed{3}$ This is the only non-trivial case. It gives update substitutions their intended semantics: If the arguments of f are equal to the ones in the update the new value as given in the update is used, otherwise the old value. Occurrences of f in $t_1, \ldots, t_n$ or t are not substituted so that the old value can be used to define the new value.

$\boxed{4}$ Logical variables are not affected by updates and x does not occur free in u. Therefore no special care is needed here.

Examples of the application of update substitutions can be found in section 7.4.

5 Structures

In this section the structures are defined for the same fixed vocabulary, called Voc, as in the preceding section. Then the notion of interpreted updates is introduced.

5.1 Definition

Definition 21. *A* structure *for the vocabulary Voc is a tuple of several objects as given below that satisfy the reachability condition.*

- A universe mapping U that maps each sort $S \in \Sigma$ to a set, the universe of S.
- A set $\mathbf{G}$ of interpretation functions. An interpretation function val_g must be such that (U, val_g) a structure of many-sorted first-order predicate logic for the first-order vocabulary induced by Voc. That means:
  - A function symbol is interpreted as a mapping between the universes according to its signature: $\mathrm{val}_g(f) : S^1 \times \ldots \times S^n \to S$ for $f \in FS$ and $\mathrm{Sign}(f) = S^1 \ \ldots\ S^n\ S$, $n \geq 1$
    $\mathrm{val}_g(f) \in U(S)$ for $f \in FS$ and $\mathrm{Sign}(f) = S$.
  - A predicate symbol is interpreted as a relation between the universes according to its signature: $\mathrm{val}_g(P) \subseteq S^1 \times \ldots \times S^n$ for $P \in PS$ and $\mathrm{Sign}(P) = S^1 \times \ldots \times S^n$.

  The interpretation of the rigid function symbols and the predicate symbols must be the same for all states of the same structure.

  This definition does not contain a set of states because states and interpretation functions are identified. val_g is used instead of $g \in \mathbf{G}$ to stress the interpretation function.
- An origin $O \in \mathbf{G}$ which is a distinguished state: The state, in which the execution of a program usually starts.
- A family $(\mathrm{exec^A}(c), \mathrm{eff^A}_c)_{c \in \underline{\mathrm{elementary}}}$ of executability conditions and effect mappings.

  An executability condition $\mathrm{exec^A}(c)$ is a subset of $\mathbf{G}$: The set of states in which c can be executed.

  An effect mapping $\mathrm{eff^A}_c$ is a mapping from $\mathrm{exec^A}(c)$ to $\mathbf{G}$: It maps a state to the state that succeeds it after the execution of c.
- A family $(\mathrm{exec^S}(c,d), \mathrm{eff^S}_{c,d})_{c,d \in \underline{\mathrm{elementary}}}$ which gives executability conditions and effect functions for the synchronous execution of two commands[11]: $\mathrm{exec^S}(c,d)$ is a subset of $\mathbf{G}$, and $\mathrm{eff^S}_{c,d}$ is a mapping from $\mathrm{exec^S}(c,d)$ to $\mathbf{G}$.

Reachability Condition A state g' is called reachable from a state g if there is a finite sequence of states $(g_0, \ldots, g_n)$ such that $g = g_0$, $g' = g_n$ and such that for every $i \in [0; n-1]$

- there is a command $c \in \underline{\mathrm{elementary}}$ satisfying $g_i \in \mathrm{exec^A}(c)$ and $g_{i+1} = \mathrm{eff^A}_c(g_i)$,
- there are two commands $c, d \in \underline{\mathrm{elementary}}$ satisfying $g_i \in \mathrm{exec^S}(c,d)$ and $g_{i+1} = \mathrm{eff^S}_{c,d}(g_i)$ or
- there is an interpreted update u (where interpreted updates are defined below) satisfying $\mathrm{val}_{g_{i+1}} = u{*}\mathrm{val}_{g_i}$.

The reachability condition requires that $\mathbf{G}$ contains all states that are reachable from the origin O.

Extensions of the Executability Conditions

Definition 22. *The mappings* $\mathrm{exec^A}(c)$ *are extended to composed commands by*

$$\mathrm{exec^A}(c) = \begin{cases} \bigcup\limits_{i=1}^{r} \mathrm{exec^A}(c[i]_1) & \text{if } c \in \underline{\mathrm{if}} \cup \underline{\mathrm{do}},\ r = opt(c) + optelse(c) \\ \mathrm{exec^A}(c[body]_1) & \text{if } c \in \underline{\mathrm{atomic}} \end{cases}$$

The mappings $\mathrm{exec^S}(c,d)$ *are extended to composed commands by*[12]

$$\mathrm{exec^S}(c,d) = \begin{cases} \bigcup\limits_{i=1}^{opt(c)} \mathrm{exec^S}(c[i]_1, d) & \text{if } c \in \underline{\mathrm{if}} \cup \underline{\mathrm{do}} \\ \bigcup\limits_{i=1}^{opt(d)} \mathrm{exec^S}(c, d[i]_1) & \text{if } d \in \underline{\mathrm{if}} \cup \underline{\mathrm{do}} \\ \mathrm{exec^S}(c[body]_1, d) & \text{if } c \in \underline{\mathrm{atomic}} \\ \mathrm{exec^S}(c, d[body]_1) & \text{if } d \in \underline{\mathrm{atomic}} \end{cases}$$

[11] It is not required that $\mathrm{exec^S}(c,d)$ and $\mathrm{eff^S}_{c,d}$ are symmetric in c and d. For example in the standard structure synchronous execution is at most possible if c is a $\underline{\mathrm{send}}$ and d a $\underline{\mathrm{receive}}$ command.

[12] If two options of the case distinction apply, it is irrelevant which one is chosen.

General Structures For simplicity the semantics as given in section 7 assumes that commands of the types $\underline{\text{else}}$, $\underline{\text{run}}$, $\underline{\text{break}}$ and $\underline{\text{end}}$ are not synchronously executable with any other command. All structures that are remotely related to Promela, in particular the standard structure, have this property since in Promela synchronous execution is only used for synchronous transfers.

For other structures the semantics is still well-defined, but in some cases in another way than intuitively expected. To cover the general case appropriately two formally easy, but notationally complex extensions would be necessary.[13]

5.2 Interpreted Updates

The interpretation of updates is not only used to give the interpretation of formulas that contain updates, but also to specify changes to interpretation functions.

Definition 23. *An* interpreted update *of the primary non-rigid function symbol f with signature $S^1 \ \dots \ S^n \ S$ is a triple (f, l, v) where $l = ()$ if $n = 0$ and $l \in U(S^1) \times \dots \times U(S^n)$ if $n > 0$, and $v \in U(S)$.*

Definition 24. *For an interpretation function val_g and an interpreted update $u = (f, l, v)$, the* application *$u * \mathrm{val}_g$ of the interpreted update u to the interpretation function val_g is the interpretation function defined by*

$$(u * \mathrm{val}_g)(s) = \begin{cases} v & \text{if } s = f,\ n = 0 \\ m \mapsto \left\{ \begin{matrix} v & \text{if } m = l \\ \mathrm{val}_g(f)(m) & \text{otherwise} \end{matrix} \right\} & \text{if } s = f,\ n > 0 \\ \mathrm{val}_g(s) & \text{otherwise} \end{cases}$$

For an interpretation function val_g and a finite sequence of updates U, the interpretation function $U * \mathrm{val}_g$ is defined inductively by

$$U * \mathrm{val}_g = \mathrm{Rest}(U) * (\mathrm{First}(U) * \mathrm{val}_g)$$

and $() * \mathrm{val}_g = \mathrm{val}_g$.

Example If $\mathrm{val}(f) : \mathbb{N} \to \mathbb{N}$, $\mathrm{val}(f)(n) = n$, $u = (f, 0, 1)$ and $v = (f, 0, 0)$ then

$$(u * \mathrm{val})(f)(n) = max\{1, n\}.$$

and $(u \cdot v) * \mathrm{val} = v * \mathrm{val} = \mathrm{val}$.

6 Truth Values

Definition 25. *The* lattice of truth values $\mathbf{B}$ *is a Boolean lattice with signature $(0, 1, \wedge, \vee, \inf, \sup, \neg)$ giving the least and the greatest element, the infimum and the supremum of two elements and a set of elements and the complement of an element, respectively.*

In this work two-valued logic[14] is used and the lattice of truth values is $\mathbf{B} = \{0, 1\}$ with the usual Boolean structure.

As usual inf and sup are also defined for the empty set: $\inf \emptyset = 1$ and $\sup \emptyset = 0$.

7 Semantics

Definition 26. *An* assignment *α for a given structure is a mapping from the logical variables to the universes such that for $S \in \Sigma$ and $x \in LV(S)$, $\alpha(x) \in U(S)$.*[15]

[13] If $\underline{\text{else}}$ commands may be synchronously executable the definition of $\mathrm{Unwind}^{\mathrm{S}}$ for $\underline{\text{if}}$ and $\underline{\text{do}}$ commands may not ignore the else option. If $\underline{\text{run}}$, $\underline{\text{break}}$ or $\underline{\text{end}}$ commands may be synchronously executable the definition of $\mathrm{rem}^{\mathrm{S}}$ must distinguish between the type of the executed command in a similar way as the definition of $\mathrm{rem}^{\mathrm{A}}$.

[14] Most of the definitions and results in this work can also be used if the truth values form any Boolean lattice.

For an assignment α, $S \in \Sigma$, $x \in LV(S)$ and $u \in U(S)$, α_x^u is the same as α with the exception that it maps x to u:

$$\alpha_x^u(y) = \begin{cases} u & \text{if } y = x \\ \alpha(y) & \text{otherwise} \end{cases}$$

The interpretation function val_g of a state g of a fixed structure

$$\big(U, \mathbf{G}, O, (\mathrm{exec}^{\mathrm{A}}(c), \mathrm{eff}^{\mathrm{A}}{}_c)_{c \in \underline{\mathrm{elementary}}}, (\mathrm{exec}^{\mathrm{S}}(c,d), \mathrm{eff}^{\mathrm{S}}{}_{c,d})_{c,d \in \underline{\mathrm{elementary}}}\big)$$

is extended to all terms, programs and formulas over the same vocabulary as the structure relative to an assignment α. This extension is denoted by $\overline{\mathrm{val}_g^\alpha}$, and the way in which it is determined by the interpretation function and the assignment is given in the following. For free terms, programs and closed formulas $\overline{\mathrm{val}_g^\alpha}$ does not depend on α and is therefore written as $\overline{\mathrm{val}_g}$.

Finally the update substitution lemma is given, which justifies the semantics of updates.

7.1 Terms

A term $t \in Terms(S)$ is interpreted as an element of $U(S)$.

Definition 27. *The* semantics of a term t *in the state g is defined by:*

- $\overline{\mathrm{val}_g^\alpha}(t) = \alpha(t)$ *if* $t \in LV$,
- $\overline{\mathrm{val}_g^\alpha}(t) = \mathrm{val}_g(f)$ *if* $t = f \in FS$
- $\overline{\mathrm{val}_g^\alpha}(t) = \mathrm{val}_g(f)\big(\overline{\mathrm{val}_g^\alpha}(t_1), \ldots, \overline{\mathrm{val}_g^\alpha}(t_n)\big)$ *if* $t = f(t_1, \ldots, t_n)$, $f \in FS$
- $\overline{\mathrm{val}_g^\alpha}(t) = \left\{\begin{matrix} \overline{\mathrm{val}_g^\alpha}(t_1) & \text{if } \overline{\mathrm{val}_g^\alpha}(C) = 1 \\ \overline{\mathrm{val}_g^\alpha}(t_2) & \text{otherwise} \end{matrix}\right\}$ *if* $t = [C?t_1 : t_2]$

Definition 28. *A term t is called* defined *in g for α if $\overline{\mathrm{val}_g^\alpha}(t) \neq \bot$ and undefined in g otherwise. "in g" is omitted if the property holds for all $g \in \mathbf{G}$.*

7.2 Programs

In this section first several support mappings are given: unwinding and remaining-program mappings. These are then used to define the semantics of programs.

7.2.1 Preliminary Definitions

Unwinding Before each execution step all processes that start with a composed command must be modified, so that only elementary commands are eligible for the indeterministic scheduling. This is done by the mappings $\mathrm{Unwind}^{\mathrm{A}}$ and $\mathrm{Unwind}^{\mathrm{S}}$.

The mapping $\mathrm{Unwind}^{\mathrm{A}}$ maps a process s, a state g and a set of commands[16] R to a set of processes. These processes arise from s by replacing the first command of s with a sequence of commands that starts with an elementary command that is asynchronously executable in the state g.

Similarly $\mathrm{Unwind}^{\mathrm{S}}$ maps two processes s and t and a state g to a set of pairs of processes that start with elementary commands that are synchronously executable in the state g.

The intuitive meaning of the unwinding functions is that replacing a process s of π with a process from $\mathrm{Unwind}^{\mathrm{A}}(s, g, R)$ or replacing two processes s and t of π with processes from $\mathrm{Unwind}^{\mathrm{S}}(s, t, g)$ precisely corresponds to the executability check and the indeterministic selection between all executable commands of Promela[17] when π is to be executed in the state g. In other words each element of the sets $\mathrm{Unwind}^{\mathrm{A}}(s, g, R)$ for all processes s of π and $\mathrm{Unwind}^{\mathrm{S}}(s, t, g)$ for all pairs (s, t) of processes of π starts a new indeterministic branch in the execution of π.

[15] The assignments are rigid, i. e. the same assignment is used for all states.

[16] See below for the meaning of R.

[17] See the specification of the Promela semantics in the two algorithms in the section semantics model in [1].

Unwinding with Respect to Asynchronous Execution To check whether a command of the type $\underline{\text{else}}$ is executable, the (asynchronous or synchronous) executability of all the other options needs to be checked. Therefore a set R of possible partners for synchronous execution in other processes occurs as an argument of Unwind^{A}.

Definition 29. *For a process s, a state g and a set of commands R, $\text{Unwind}^{\text{A}}(s,g,R)$ is defined by structural induction on s:*

$$\text{Unwind}^{\text{A}}(s,g,R) = \begin{cases} \{s';\text{Rest}(s) \mid s' \in \text{Unwind}^{\text{A}}(s_1,g,R)\} & \text{if Length}(s) > 1 \ \boxed{1} \\ \{c\} & \text{if } s = c,\ c \in \underline{\text{elementary}}, \\ & \quad g \in \text{exec}^{\text{A}}(c) \ \boxed{2} \\ \emptyset & \text{if } s = c,\ c \in \underline{\text{elementary}}, \\ & \quad g \notin \text{exec}^{\text{A}}(c) \ \boxed{3} \\ \bigcup_{i=1}^{opt(c)} \text{Unwind}^{\text{A}}(c[i],g,R) \cup \overbrace{\text{else}(c,g,R)}^{\boxed{4}} & \text{if } s = c,\ c \in \underline{\text{if}} \ \boxed{5} \\ \left\{ s'; \overbrace{\text{BT } c}^{\boxed{6}} \,\middle|\, s' \in \bigcup_{i=1}^{opt(c)} \text{Unwind}^{\text{A}}(c[i],g,R) \cup \text{else}(c,g,R) \right\} & \text{if } s = c,\ c \in \underline{\text{do}} \\ \bigcup_{b' \in \text{Unwind}^{\text{A}}(b,g,R)} \text{Unwind}^{\text{A}}(\text{ATOMIC}\{b'\},g,R) & \text{if } s = c,\ c \in \underline{\text{atomic}},\ b = c[body], \\ & \quad b_1 \in \underline{\text{if}} \cup \underline{\text{do}} \ \boxed{7} \\ \text{Unwind}^{\text{A}}\big(\overset{\boxed{8}}{b_1}; \overbrace{\text{ECAtomic}(\text{Rest}(b))}^{\boxed{9}}, g, R\big) & \text{if } s = c,\ c \in \underline{\text{atomic}},\ b = c[body], \\ & \quad b_1 \in \underline{\text{elementary}} \cup \underline{\text{atomic}} \end{cases}$$

where

$$\text{else}(c,g,R) = \begin{cases} \text{Unwind}^{\text{A}}(c[else],g,R) & \text{if } optelse(c) = 1,\ \overbrace{\text{Unwind}^{\text{A}}(c[i],g,R) = \emptyset}^{\boxed{10}}, \\ & \quad \overbrace{\text{Unwind}^{\text{S}}(c[i],r,g) = \emptyset}^{\boxed{11}} \text{ for } r \in R, i \in [1;opt(c)] \\ \emptyset & \text{otherwise} \end{cases}$$

$\boxed{1}$ The first command of the sequence is unwound recursively. The rest is kept.

$\boxed{2}$ For executable elementary commands the recursion terminates. This must happen eventually since only a finite depth of nesting of composed commands is possible.

$\boxed{3}$ If an elementary command is not executable the recursion terminates, too, but no sequences are returned.

$\boxed{4}$ This unwinds the else option if no non-else option is executable. It is defined below.

$\boxed{5}$ All options are unwound separately and the results are united.

$\boxed{6}$ The only difference between commands of the types $\underline{\text{if}}$ and $\underline{\text{do}}$ is that a copy of the $\underline{\text{do}}$ loop is kept. The label BT marks a loop that must be skipped if a $\underline{\text{break}}$ is executed.

$\boxed{7}$ The first command of the atomic sequence is a selection. Therefore the body is unwound and replaced with the results of the unwinding. Then the unwinding continues.

$\boxed{8}$ The first command of the atomic sequence is no selection and is extracted from the atomic sequence and unwound.

$\boxed{9}$ The rest of the sequence is marked with the EC label to ensure that the execution continues with the atomic sequence if the first command has been executed. If the body has been completely executed Rest(b) is empty and so is ECAtomic(Rest(b)).

$\boxed{10}$ This means that none of the non-else options is asynchronously executable.

$\boxed{11}$ This means that none of the non-else options is synchronously executable with a partner from R in the second position. This is the only reason why the argument R is needed. Unwind^{S} is defined below.

Unwinding with Respect to Synchronous Execution The definition of Unwind^{S} is essentially the same as that of Unwind^{A} with the following changes:

- The synchronous executability of two commands is checked instead of asynchronous executability.
- Pairs of processes are returned because there are two arguments that must be unwound.
- There are two cases for each composed command because there are two arguments that may be a composed command.
- The else options can be ignored, because their first command ELSE cannot be executed synchronously.[18]

Definition 30. *For two processes s and t and a state g, $\mathrm{Unwind}^{\mathrm{S}}(s,t,g)$ is defined by structural induction on s and t:*[19]

$$\mathrm{Unwind}^{\mathrm{S}}(s,t,g) = \begin{cases} \big\{(s';\mathrm{Rest}(s)\ ,\ t';\mathrm{Rest}(s'))|(s',t') \in \mathrm{Unwind}^{\mathrm{S}}(s_1,t_1,g)\big\} & \text{if } \mathrm{Length}(s)>1 \text{ or } \mathrm{Length}(t)>1 \\ \{(c,d)\} & \text{if } s=c,\ t=d,\ c,d \in \underline{\text{elementary}} \\ & \quad g \in \mathrm{exec}^{\mathrm{S}}(c,d) \\ \emptyset & \text{if } s=c,\ t=d,\ c,d \in \underline{\text{elementary}} \\ & \quad g \notin \mathrm{exec}^{\mathrm{S}}(c,d) \\ \bigcup_{i=1}^{opt(c)} \mathrm{Unwind}^{\mathrm{S}}(c[i],t,g) & \text{if } s=c,\ c\in \underline{\text{if}} \\ \bigcup_{i=1}^{opt(c)} \mathrm{Unwind}^{\mathrm{S}}(s,c[i],g) & \text{if } t=c,\ c\in \underline{\text{if}} \\ \bigcup_{i=1}^{opt(c)} \big\{(s';\mathrm{BT}\ c\ ,\ t')|(s';t') \in \mathrm{Unwind}^{\mathrm{S}}(c[i],t,g)\big\} & \text{if } s=c,\ c\in \underline{\text{do}} \\ \bigcup_{i=1}^{opt(c)} \big\{(s'\ ,\ t';\mathrm{BT}\ c)|(s',t') \in \mathrm{Unwind}^{\mathrm{S}}(s,c[i],g)\big\} & \text{if } t=c,\ c\in \underline{\text{do}} \\ \bigcup_{b'\in \mathrm{Unwind}^{\mathrm{S}}(b,t,g)} \mathrm{Unwind}^{\mathrm{S}}(\mathrm{ATOMIC}\{b'\},t,g) & \text{if } s=c,\ c\in\underline{\text{atomic}},\ b=c[body], \\ & \quad b_1 \in \underline{\text{if}} \cup \underline{\text{do}} \\ \bigcup_{b'\in \mathrm{Unwind}^{\mathrm{S}}(s,b,g)} \mathrm{Unwind}^{\mathrm{S}}(s,\mathrm{ATOMIC}\{b'\},g) & \text{if } t=c,\ c\in\underline{\text{atomic}},\ b=c[body], \\ & \quad b_1 \in \underline{\text{if}} \cup \underline{\text{do}} \\ \mathrm{Unwind}^{\mathrm{S}}\big(b_1;\mathrm{ECAtomic}(\mathrm{Rest}(b))\ ,\ t\ ,\ g\big) & \text{if } s=c,\ c\in\underline{\text{atomic}},\ b=c[body], \\ & \quad b_1 \in \underline{\text{elementary}} \cup \underline{\text{atomic}} \\ \mathrm{Unwind}^{\mathrm{S}}\big(s\ ,\ b_1;\mathrm{ECAtomic}(\mathrm{Rest}(b))\ ,\ g\big) & \text{if } t=c,\ c\in\underline{\text{atomic}},\ b=c[body], \\ & \quad b_1 \in \underline{\text{elementary}} \cup \underline{\text{atomic}} \end{cases}$$

Examples The idea behind the unwinding functions is illustrated by the following examples: Let $r_1,\dots,r_3$ be elementary commands, $s_1,\dots,s_5$ be command sequences and

$c =$ DO
 :: $r_1; s_1$
 :: $r_2; s_2$
 :: s_3
 :: ELSE; s_4
OD

And let $g \in \mathrm{exec}^{\mathrm{A}}(r_1)$, $g \notin \mathrm{exec}^{\mathrm{A}}(r_2)$, $g \notin \mathrm{exec}^{\mathrm{S}}(r_1,r_3)$ and $g \in \mathrm{exec}^{\mathrm{S}}(r_2,r_3)$.

Then

$$\mathrm{Unwind}^{\mathrm{A}}(c;s_5,g,\{r_3\}) = \big\{r_1;s_1;\mathrm{BT}\ c;s_5\big\} \cup \big\{s;\mathrm{BT}\ c;s_5 | s \in \mathrm{Unwind}^{\mathrm{A}}(s_3,g,\{r_3\})\big\}$$

[18] See the remark on general structures above.

[19] If two options of the case distinction apply, it is irrelevant which one is chosen.

and

$$\mathrm{Unwind}^{\mathrm{S}}(c;s_5,r_3,g)=\big\{(r_2;s_2;\mathrm{BT}\ c;s_5\ ,\ r_3)\big\}\cup\big\{(s;\mathrm{BT}\ c;s_5\ ,\ t)|(s,t)\in\mathrm{Unwind}^{\mathrm{S}}(s_3,r_3,g)\big\}$$

$\mathrm{Unwind}^{\mathrm{A}}$: The first option of c is asynchronously executable leading to the execution of the command sequence $r_1;s_1;\mathrm{BT}\ c;s_5$. The second option is not asynchronously executable and ignored. The executability of the third option is not clear and $\mathrm{First}(s_3)$ may be a composed command; therefore the recursive call is given explicitly; BT c and the commands of s_5 are appended to each result of this recursive call. The else option is ignored because there is an executable non-else option. Even if no non-else option were asynchronously executable the else option would still be ignored because the second option is synchronously executable with r_3.

$\mathrm{Unwind}^{\mathrm{S}}$: r_1 and r_3 are not synchronously executable. r_2 and r_3 are synchronously executable leading to the execution of the command sequences $r_2;s_2;\mathrm{BT}\ c;s_5$ and r_3. The synchronous executability of s_3 and r_3 is checked by the recursive call. The else option is ignored completely.

The following examples illustrate the complicated unwinding of composed commands within atomic sequences where additionally $g\in\mathrm{exec}^{\mathrm{A}}(r_3)$:

$$\begin{aligned}&\mathrm{Unwind}^{\mathrm{A}}\big(\mathrm{ATOMIC}\{\mathrm{IF}\ ::\ r_1;s_1\ ::r_2;s_2\ ::r_3;s_3\ \mathrm{FI};\ s_4\}\ ,\ g\ ,\ \emptyset\big)\\&=\mathrm{Unwind}^{\mathrm{A}}\big(\mathrm{ATOMIC}\{r_1;s_1;\ s_4\}\ ,\ g\ ,\ \emptyset\big)\cup\mathrm{Unwind}^{\mathrm{A}}\big(\mathrm{ATOMIC}\{r_3;s_3;\ s_4\}\ ,\ g\ ,\ \emptyset\big)\\&=\mathrm{Unwind}^{\mathrm{A}}\big(r_1;\mathrm{EC\ ATOMIC}\{s_1;\ s_4\}\ ,\ g\ ,\ \emptyset\big)\cup\mathrm{Unwind}^{\mathrm{A}}\big(r_3;\mathrm{EC\ ATOMIC}\{s_3;\ s_4\}\ ,\ g\ ,\ \emptyset\big)\\&=\big\{r_1;\mathrm{EC\ ATOMIC}\{s_1;\ s_4\}\ ,\ r_3;\mathrm{EC\ ATOMIC}\{s_3;\ s_4\}\big\}\end{aligned}$$

and[20]

$$\begin{aligned}&\mathrm{Unwind}^{\mathrm{A}}\big(\mathrm{ATOMIC}\{\mathrm{ATOMIC}\{r_1;s_1\};s_2\}\ ,\ g\ ,\ \emptyset\big)\\&=\mathrm{Unwind}^{\mathrm{A}}\big(\mathrm{ATOMIC}\{r_1;s_1\};\mathrm{EC\ ATOMIC}\{s_2\}\ ,\ g\ ,\ \emptyset\big)\\&=\mathrm{Unwind}^{\mathrm{A}}\big(r_1;\mathrm{EC\ ATOMIC}\{s_1\};\mathrm{EC\ ATOMIC}\{s_2\}\ ,\ g\ ,\ \emptyset\big)\\&=\big\{r_1;\mathrm{EC\ ATOMIC}\{s_1\};\mathrm{EC\ ATOMIC}\{s_2\}\big\}\end{aligned}$$

Elementary Properties

Lemma 1. *For all command sequences s, all states g and all sets of commands R:*

- $\mathrm{Unwind}^{\mathrm{A}}$ *is idempotent in the following sense:* $\mathrm{Unwind}^{\mathrm{A}}(s,g,R)=\bigcup_{s'\in\mathrm{Unwind}^{\mathrm{A}}(s,g,R)}\mathrm{Unwind}^{\mathrm{A}}(s',g,R)$
- $|\mathrm{Unwind}^{\mathrm{A}}(s,g,R)|<\infty$
- $s'\in\mathrm{Unwind}^{\mathrm{A}}(s,g,R)=$ *implies* $s'_1\in\underline{\mathrm{elementary}}$ *and* $g\in\mathrm{exec}^{\mathrm{A}}(s'_1)$
- $s_1\in\underline{\mathrm{elementary}}$ *implies* $\mathrm{Unwind}^{\mathrm{A}}(s,g,R)\subseteq\{s\}$

Similarly, for all command sequences s,t and all states g:

- $\mathrm{Unwind}^{\mathrm{S}}(s,t,g)=\bigcup_{(s',t')\in\mathrm{Unwind}^{\mathrm{S}}(s,t,g)}\mathrm{Unwind}^{\mathrm{S}}(s',t',g)$
- $|\mathrm{Unwind}^{\mathrm{A}}(s,t,g)|<\infty$
- $(s',t')\in\mathrm{Unwind}^{\mathrm{S}}(s,t,g)=$ *implies* $s'_1,t'_1\in\underline{\mathrm{elementary}}$ *and* $g\in\mathrm{exec}^{\mathrm{S}}(s'_1,t'_1)$
- $s_1,t_1\in\underline{\mathrm{elementary}}$ *implies* $\mathrm{Unwind}^{\mathrm{S}}(s,t,g)\subseteq\{(s,t)\}$.

Proof. The proofs are simple applications of the definitions.

The finiteness properties hold because if and do commands may have only finitely many options and composed commands may be nested only finitely many times. Formally, induction on the depth of nesting of composed commands is used; for a process s this is $\mathit{depth}(\mathrm{First}(s))$ which is defined by

$$\mathit{depth}(c)=\begin{cases}1+\mathit{depth}(\mathrm{First}(c[\mathit{body}])) & \text{if } c\in\underline{\mathrm{atomic}}\\ 1+\max\limits_{i\in[1;\mathit{opt}(c)+\mathit{optelse}(c)]}\mathit{depth}(\mathrm{First}(c[i])) & \text{if } c\in\underline{\mathrm{if}}\cup\underline{\mathrm{do}}\\ 0 & \text{otherwise}\end{cases}$$

in the synchronous case $\mathit{depth}(\mathrm{First}(s))+\mathit{depth}(\mathrm{First}(t))$ is used. □

[20] Of course it is useless to nest atomic sequence.

The Remaining Program after Asynchronous Execution $\mathrm{rem}^{\mathrm{A}}(\pi, i)$ denotes the program that remains to be executed after the first command of π_i has been executed asynchronously.

Definition 31. *Let π be a program and $i \in [1;\mathrm{Length}(\pi)]$ where $c = \mathrm{First}(\pi_i) \in \underline{\mathrm{elementary}}$. Then*

$$\mathrm{rem}^{\mathrm{A}}(\pi, i) = \begin{cases} \overset{\boxed{1}}{\pi^i}(\overbrace{\mathrm{Dropbt}}^{\boxed{2}}(\mathrm{Rest}(\pi_i))) & \text{if } c \in \underline{\mathrm{assignment}} \cup \underline{\mathrm{expression}} \cup \underline{\mathrm{send}} \cup \underline{\mathrm{receive}} \cup \underline{\mathrm{else}}\ \boxed{3} \\ \mathrm{Remove}(\pi, \{i\}) & \text{if } c \in \underline{\mathrm{end}}\ \boxed{4} \\ \pi^i(\mathit{break}(\mathrm{Rest}(\pi_i))) & \text{if } c \in \underline{\mathrm{break}}\ \boxed{5} \\ \sigma\Big(\pi^i\big(\mathrm{Dropbt}(\mathrm{Rest}(\pi_i))\big) \mid \mathrm{Body}(\mathrm{PT})\Big) & \text{if } c \in \underline{\mathrm{run}},\ \mathit{Proc}(c) = \mathrm{PT}\ \boxed{6} \end{cases}$$

where

- *$\mathit{break}(s)$ denotes the part of s that follows after the first command in s that is labelled* BT*:*

$$\mathit{break}(s) = \begin{cases} s & \text{if } s = () \text{ or } s \in \underline{\mathrm{end}}\ \boxed{7} \\ \mathrm{Rest}(s) & \text{if } \mathrm{Label}(s_1) = \mathrm{BT}\ \boxed{8} \\ \mathrm{Label}(s_1)\ \mathrm{ATOMIC}\{\mathit{break}(b)\};\mathrm{Rest}(s) & \text{if } s_1 \in \underline{\mathrm{atomic}},\ b = s_1[\mathit{body}],\ \overbrace{\mathit{break}(b) \neq ()}^{\boxed{9}}\ \boxed{10} \\ \mathit{break}(\mathrm{Rest}(s)) & \text{otherwise}\ \boxed{11} \end{cases}$$

and

- *σ is the mapping that replaces every occurrence of the function symbol nextpid in its argument with $\mathit{nextpid} - c_1$* $\boxed{12}$.

The Remaining Program after Synchronous Execution $\mathrm{rem}^{\mathrm{S}}(\pi, i, j)$ denotes the program that remains to be executed after the first commands of π_i and π_j have been executed synchronously.

Definition 32. *Let π be a program, $i, j \in \mathrm{Length}(\pi)$ and $i \neq j$ where $\mathrm{First}(\pi_i), \mathrm{First}(\pi_j) \in \underline{\mathrm{elementary}}$. Then*

$$\mathrm{rem}^{\mathrm{S}}(\pi, i, j) = \Big(\pi^i\big(\overbrace{\mathrm{Dropec}(\mathrm{Dropbt}(\mathrm{Rest}(\pi_i)))}^{\boxed{13}}\big)\Big)^j\big(\mathrm{Dropbt}(\mathrm{Rest}(\pi_j))\big)\ \boxed{14}$$

This definition contains a crucial scheduling decision. If one or both executed commands are part of an atomic sequence it has to be defined which if any process has exclusive control after the execution. The occurrence of the Dropec mapping entails that possible exclusive control of π_i is lost. If $\mathrm{Rest}(\pi_j)$ starts with an EC label, π_j will keep or gain exclusive control. This behavior is the same as in Promela.

$\boxed{1}$ By the definition of $\pi^i(s)$ the i-th process is removed if $s = ()$.

$\boxed{2}$ If the last command of an option of a <u>do</u> command is executed, the BT label of the following <u>do</u> command is dropped.

$\boxed{3}$ In most cases the executed command is simply dropped.

$\boxed{4}$ The terminated process is removed.

$\boxed{5}$ A <u>break</u> command requires to skip everything until after the next BT label.

$\boxed{6}$ As in the first case, but additionally the body of the instantiated process type is added and all occurrences of local variables are modified.

$\boxed{7}$ This case applies if the whole sequence of commands has been searched without the occurrence of a BT label. This happens if the <u>break</u> occurred outside of any <u>do</u> command or if the body of an atomic sequence is searched.

$\boxed{8}$ If the first command is labelled BT it is dropped and the recursion terminates.

$\boxed{9}$ If this condition is not satisfied there is no BT label in the body of the <u>atomic</u> command and the search continues after the atomic sequence.

$\boxed{10}$ If the <u>break</u> command occurs inside the body of an <u>atomic</u> command it is extracted by the $\mathrm{Unwind}^{\mathrm{A}}$ function. Therefore the body must be searched for the target of the BREAK command.

$\boxed{11}$ All commands are dropped until a BT label or the end of the command sequence is encountered.

$\boxed{12}$ According to the definition of process types the first argument in all references to local variables and the second argument of the <u>end</u> command are initially *nextpid*, the next process ID. σ decrements these arguments by 1. And if[21] instantiating a new process increments the value of *nextpid* by 1, the application of σ ensures that all references to *nextpid* in the declaration of the process type are always interpreted as the process ID of the containing process.

References to global variables are unchanged by σ.

$\boxed{13}$ Possible exclusive control of π_i is lost. If the executed command was the last command of an option of a <u>do</u> command, the leading BT label is dropped.

$\boxed{14}$ The synchronously executed commands are simply removed.[22]

These definitions guarantee that the order of the processes in π is always the order in which the processes have been created.

Example The following example illustrates the two most complicated cases of the application of $\mathrm{rem^A}$.

Let
$\pi = \mathrm{RUN\ PT_3};\ X_1^l;\ \mathrm{END}(\mathrm{PT_1}, \mathit{nextpid} - c_1 - c_1) \mid$
$\quad \mathrm{BREAK};\ \mathrm{EC\ ATOMIC}\{s_1;\ \mathrm{BT}\ c;\ s_2\};\ \mathrm{END}(\mathrm{PT_2}, \mathit{nextpid} - c_1)$
where s_1 and s_2 are command sequences, $\mathit{Body}(\mathrm{PT_3}) = s_3; \mathrm{END}(\mathrm{PT_3}, \mathit{nextpid})$ and $c \in$ <u>do</u>.

Then
$\mathrm{rem^A}(\pi, 1) = X_1^l;\ \mathrm{END}(\mathrm{PT_1}, \mathit{nextpid} - c_1 - c_1 - c_1) \mid$
$\qquad \mathrm{BREAK};\ \mathrm{EC\ ATOMIC}\{s_1;\ \mathrm{BT}\ c;\ s_2\};\ \mathrm{END}(\mathrm{PT_2}, \mathit{nextpid} - c_1 - c_1) \mid$
$\qquad s_3;\ \mathrm{END}(\mathrm{PT_3}, \mathit{nextpid} - c_1)$
and
$\mathrm{rem^A}(\pi, 2) = \mathrm{RUN\ PT_3};\ X_1^l;\ \mathrm{END}(\mathrm{PT_1}, \mathit{nextpid} - c_1 - c_1) \mid$
$\qquad \mathrm{EC\ ATOMIC}\{s_2\};\ \mathrm{END}(\mathrm{PT_2}, \mathit{nextpid} - c_1)$.

In both cases the local program variable abbreviation X_1^l remains unchanged. But this is not trivial in the first case: Formally, the abbreviation must be eliminated before σ is applied and re-introduced afterwards, but it can easily be shown that no such abbreviations of local and global program variables are ever changed.

7.2.2 Definition

Definition 33. *A* trace *is a non-empty sequence over the set of labelled states where the possible labels are ϵ,* <u>termination</u> *and* <u>timeout</u>. *For a state g and a label L let $L\ g$ denote the labelled state, and let* $\mathrm{Label}(L\ g) = L$ *and* $\mathrm{RemLabel}(L\ g) = g$.

The labels <u>termination</u> and <u>timeout</u> are used to distinguish between normal and abnormal program termination. In DLTP (as in Promela) all possible run-time errors are prevented by the executability conditions; therefore there are no execution aborts, but only timeouts.

A program π is interpreted as a set of traces: The indeterministically possible traces if π is executed in the state g. The semantics of a program with an empty tag is defined relative to the semantics of the same program with all possible non-empty tags.

Programs with Non-empty Tags

Definition 34. *For $i, j \in [1; n]$, $i \neq j$ the* semantics of programs with non-empty tags *in the state g is defined in the following way:*

- $\overline{\mathrm{val}}_g(i:\ \pi)$ *is the set of traces that start with the execution of an asynchronous command from π_i:*

$$\overline{\mathrm{val}}_g(i:\ \pi) = \Big\{(g)\cdot h \mid \overbrace{s \in \mathrm{Unwind^A}(\pi_i, g, R_i)}^{\boxed{1}},\ \overbrace{g' = \mathrm{eff^A}_{s_1}(g)}^{\boxed{2}},\ \overbrace{h \in \overline{\mathrm{val}}_{g'}\big(\mathrm{rem^A}(\pi^i(s), i)\big)}^{\boxed{3}}\Big\}$$

---

[21] Of course that depends on the structure.

[22] This makes sense if <u>run</u>, <u>break</u> and <u>end</u> commands are never synchronously executable. See the remark on general structures above.

where

$$R_i = \overbrace{\{\mathrm{First}(\pi_j) | j \in [1;n] \setminus \{i\}\}}^{\boxed{4}}$$

- $\overline{\mathrm{val}}_g(i, j : \pi)$ *is the set of traces that start with the synchronous execution of commands from* π_i *and* π_j*:*

$$\overline{\mathrm{val}}_g(i, j : \pi) = \Big\{ (g) \cdot h \big| \overbrace{(s,t) \in \mathrm{Unwind}^{\mathrm{S}}(\pi_i, \pi_j, g)}^{\boxed{5}}, \overbrace{g' = \mathrm{eff}^{\mathrm{S}}{}_{s_1,t_1}(g)}^{\boxed{6}}, \overbrace{h \in \overline{\mathrm{val}_{g'}}\big(\mathrm{rem}^{\mathrm{S}}((\pi^i(s))^j(t), i, j)\big)}^{\boxed{7}} \Big\}$$

$\boxed{1}$ s runs over all processes that π_i can be replaced with to indeterministically determine the next command.

$\boxed{2}$ The first command of s is elementary and asynchronously executable. Its effect is applied to g giving the next state g'.

$\boxed{3}$ π_i is replaced with s and then the remaining program after execution of s_1 is formed. h runs over all traces the remaining program can result in.

$\boxed{4}$ This is the set of first commands of the other processes; these may provide the partner for the synchronous execution of a command in π_i.

$\boxed{5}$ s and t run over all pairs of processes that π_i and π_j can be replaced with to indeterministically determine the next command.

$\boxed{6}$ The first commands of s and t are elementary and synchronously executable. Their effect is applied to g giving the next state g'.

$\boxed{7}$ π_i and π_j are replaced with s and t and then the remaining program after execution of s_1 and t_1 is formed. h runs over all traces the remaining program can result in.

$\overline{\mathrm{val}}_g(i : \pi)$ is empty if $g \notin \mathrm{exec}^{\mathrm{A}}(\mathrm{First}(\pi_i))$, and similarly for the synchronous case.

Programs with Empty Tags

Definition 35. *For a program* $\pi = \pi_1 | \ldots | \pi_n$*, a state* g *and* $i \in [1;n]$ *let*

$$T_i = \overline{\mathrm{val}}_g(i : \pi) \cup \bigcup_{\substack{j=1 \\ j \neq i}}^{n} \overline{\mathrm{val}}_g(i, j : \pi)$$

be the set of traces that start with the asynchronous or synchronous execution of a command from π_i $\boxed{1}$. *Then the* semantics of a program with an empty tag *is defined by:*

$$\overline{\mathrm{val}}_g(\pi) = \begin{cases} \{(\underline{\mathrm{termination}}\ g)\} & \text{if } \pi = \epsilon \ \boxed{2} \\ T_i & \text{if } M = \{j \in [1;n] | \mathrm{First}(\pi_j) \in \underline{\mathrm{end}}\} \neq \emptyset,\ \overbrace{i = \min\ M}^{\boxed{3}}\ \boxed{4} \\ T_i & \text{if } \mathrm{First}(\pi_j) \notin \underline{\mathrm{end}} \text{ for } j \in [1;n],\ \overbrace{\mathrm{Label}(\mathrm{First}(\pi_i)) = \mathrm{EC}}^{\boxed{5}},\ \overbrace{T_i \neq \emptyset}^{\boxed{6}} \\ \overline{\mathrm{val}}_g\overbrace{(\mathrm{Dropec}(\pi))}^{\boxed{7}} & \text{if } \mathrm{First}(\pi_j) \notin \underline{\mathrm{end}} \text{ for } j \in [1;n],\ \mathrm{Label}(\mathrm{First}(\pi_i)) = \mathrm{EC},\ \overbrace{T_i = \emptyset}^{\boxed{8}} \\ \{\overbrace{(\underline{\mathrm{timeout}}\ g)}^{\boxed{9}}\} & \text{if } \pi \neq \epsilon,\ \overbrace{\bigcup_{i=1}^{n} T_i = \emptyset}^{\boxed{10}} \\ \bigcup_{i=1}^{n} T_i & \text{otherwise } \boxed{11} \end{cases}$$

$\boxed{1}$ The synchronous executability of two commands does not need to be symmetric in its arguments. The first command is scheduled for synchronous execution; then it is checked whether a second command is available that makes the pair synchronously executable. Therefore a process with exclusive control can provide the first, but not the second command of such a pair.

$\boxed{2}$ The program terminates.

$\boxed{3}$ After the synchronous execution of two commands there may be two processes that have terminated; then the variables of the process that has been started first are reset first.

$\boxed{4}$ If a process has terminated the indeterministic scheduling is overruled and the $\underline{\text{end}}$ command destructs the process.

$\boxed{5}$ π_i has exclusive control ...

$\boxed{6}$ ... and one of its next commands can be executed.

$\boxed{7}$ π_i loses exclusive control ...

$\boxed{8}$ ... because First(π_i) cannot be executed.

$\boxed{9}$ A timeout occurs ...

$\boxed{10}$ ... because no command is executable.

$\boxed{11}$ A command from any process is chosen to be executed next.

Program Termination

Definition 36. *A program π* terminates *from g if all traces in $\overline{\text{val}}_g(\pi)$ are finite and have a last element that is labelled with* $\underline{\text{termination}}$*.*

$\overline{\text{val}}_g(\pi)$ is defined recursively. If π terminates from g (or timeouts occur), the recursion is well-founded. This leads to finite sets of finite traces. But in general the recursion does not need to end. Therefore a trace in $\overline{\text{val}}_g(\pi)$ is a recursively defined potentially infinite sequence of states.

Intermediate and Final States

Definition 37. *For a trace $t = (t_i)_{i\in D}$*

$$IntMed(t) = \big\{\text{RemLabel}(t_i) | i \in D\big\}$$

denotes the set of intermediate states *of t. And*

$$Fin(t) = \begin{cases} \{\text{RemLabel}(\text{Last}(t))\} & \text{if Label(Last}(t)) = \underline{\text{termination}} \\ \emptyset & \text{otherwise} \end{cases}$$

denotes the set of final states *of t.*

Examples Let $r_i \in \underline{\text{receive}}$[23] for $i \in [1;4]$, $\text{exec}^{\text{A}}(r_i) = \mathbf{G}$ for $i \in [1;3]$, $\text{exec}^{\text{A}}(r_4) = \{g_1, g_2\}$, $\text{exec}^{\text{S}}(r_i, r_j) = \text{exec}^{\text{S}}(r_i, \text{BREAK}) = \emptyset$ for $i, j \in [1,4]$ and $\text{eff}^{\text{A}}{}_{r_i}(h) = g_i$ for $i \in [1;4]$, all $h \in \mathbf{G}$ and pairwise different $g, g_1, \dots, g_4$.

If $\pi =$ DO :: r_1 :: r_2 :: BREAK OD |
EC r_3

then

$$\overline{\text{val}}_g(\pi) = \overline{\text{val}}_g(2:\ \pi) = \big\{L\big((g, g_3)\cdot h\big)|\ h \in S\big\}$$

where L labels the last state of a finite sequence with $\underline{\text{termination}}$ and leaves infinite sequences unchanged, and S denotes the set of all (empty, finite or infinite) sequences over the set $\{g_1, g_2\}$.

If $\pi =$ IF :: r_1; r_2 :: r_3 FI |
EC r_4

Then

$$\overline{\text{val}}_g(2:\ \pi) = \emptyset$$

and

$$\overline{\text{val}}_g(\pi) = \overline{\text{val}}_g(1:\ \pi) = \{(g, g_1, g_2, \underline{\text{termination}}\ g_4), (g, g_1, g_4, \underline{\text{termination}}\ g_2), (g, \underline{\text{timeout}}\ g_3)\}$$

[23] $\underline{\text{receive}}$ is used because such commands have both non-trivial executability and effect, and do not require any special treatment.

7.3 Formulas

Definition 38. *The* semantics of a sequence of updates *is the sequence of interpreted updates defined by replacing all terms with their semantics:*

- $\overline{\mathrm{val}_g^\alpha}(()) = ()$
- $\overline{\mathrm{val}_g^\alpha}\big((f,(t_1,\dots,t_n),t)\big) = \Big(f, \big(\overline{\mathrm{val}_g^\alpha}(t_1),\dots,\overline{\mathrm{val}_g^\alpha}(t_n)\big), \overline{\mathrm{val}_g^\alpha}(t)\Big)$
- $\overline{\mathrm{val}_g^\alpha}(u \cdot U) = \overline{\mathrm{val}_g^\alpha}(u) \cdot \overline{\mathrm{val}_g^\alpha}(U)$

State maps a state, an assignment and a finite sequence of updates to the state that is reached after applying the corresponding interpreted updates:

$$State(g,\alpha,U) = \overline{\mathrm{val}_g^\alpha}(U) * \mathrm{val}_g$$

A formula F is interpreted as an element of $\mathbf{B}$, its truth value.

Definition 39. *The* semantics of a formula F *in the state g under the assignment α is defined by:*

- $\overline{\mathrm{val}_g^\alpha}(true) = 1$ *and*
  $\overline{\mathrm{val}_g^\alpha}(false) = 0$
- $\overline{\mathrm{val}_g^\alpha}(P(t_1,\dots,t_n)) = 1$ *if* $(\overline{\mathrm{val}_g^\alpha}(t_1),\dots,\overline{\mathrm{val}_g^\alpha}(t_n)) \in \mathrm{val}_g(P)$ *and*
  $\overline{\mathrm{val}_g^\alpha}(P(t_1,\dots,t_n)) = 0$ *if* $(\overline{\mathrm{val}_g^\alpha}(t_1),\dots,\overline{\mathrm{val}_g^\alpha}(t_n)) \notin \mathrm{val}_g(P)$ *for $P \in PS$*
- $\overline{\mathrm{val}_g^\alpha}(\neg F) = \neg\overline{\mathrm{val}_g^\alpha}(F)$,
  $\overline{\mathrm{val}_g^\alpha}(F \wedge G) = \overline{\mathrm{val}_g^\alpha}(F) \wedge \overline{\mathrm{val}_g^\alpha}(G)$ *and*
  $\overline{\mathrm{val}_g^\alpha}(F \vee G) = \overline{\mathrm{val}_g^\alpha}(F) \vee \overline{\mathrm{val}_g^\alpha}(G)$
- $\overline{\mathrm{val}_g^\alpha}(\forall x\ F) = \inf\limits_{u \in U(S)} \overline{\mathrm{val}_g^{\alpha_x^u}}(F)$ *and*
  $\overline{\mathrm{val}_g^\alpha}(\exists x\ F) = \sup\limits_{u \in U(S)} \overline{\mathrm{val}_g^{\alpha_x^u}}(F)$ *for $x \in LV(S)$* $\boxed{1}$
- $\overline{\mathrm{val}_g^\alpha}(M(\pi)F) = \mathrm{SemModal}(M, \overline{\mathrm{val}_g}(\pi), \alpha, F)$ *where for a modal operator $M \in Oper$, a set of traces T, an assignment α and a formula F:*

$$\mathrm{SemModal}(M,T,\alpha,F) = \begin{cases} \inf\limits_{t\in T, h \in Fin(t)} \overline{\mathrm{val}_h^\alpha}(F) & \text{if } M = [\,]\ \boxed{2} \\ \sup\limits_{t\in T, h \in Fin(t)} \overline{\mathrm{val}_h^\alpha}(F) & \text{if } M = \langle\rangle\ \boxed{3} \\ \inf\limits_{t\in T} \inf\limits_{h \in IntMed(t)} \overline{\mathrm{val}_h^\alpha}(F) & \text{if } M = [[\,]]\ \boxed{4} \\ \sup\limits_{t\in T} \sup\limits_{h \in IntMed(t)} \overline{\mathrm{val}_h^\alpha}(F) & \text{if } M = \langle\langle\rangle\rangle\ \boxed{5} \\ \inf\limits_{t\in T} \sup\limits_{h \in IntMed(t)} \overline{\mathrm{val}_h^\alpha}(F) & \text{if } M = [\langle\rangle]\ \boxed{6} \\ \sup\limits_{t\in T} \inf\limits_{h \in IntMed(t)} \overline{\mathrm{val}_h^\alpha}(F) & \text{if } M = \langle[\,]\rangle\ \boxed{7} \end{cases}$$

- $\overline{\mathrm{val}_g^\alpha}(u\ F) = \overline{\mathrm{val}_{State(g,\alpha,u)}^\alpha}(F)$ $\boxed{8}$

$\boxed{1}$ These are the usual interpretations of first-order quantifiers. inf corresponds to universal and sup to existential quantification.

$\boxed{2}$ "for all final states" or "for all traces t and for all final states of t"

$\boxed{3}$ "for some final state" or "for some trace t and for some final state of t"

$\boxed{4}$ "for all traces t and for all states in t"

$\boxed{5}$ "for some trace t and for some state in t"

$\boxed{6}$ "for all traces t and for some state in t"

$\boxed{7}$ "for some trace t and for all states in t"

$\boxed{8}$ F is not interpreted in g, but in the state that is reached from g by applying the interpretation of the update u.

Examples Let a vocabulary and a structure be given such f is a unary operator on the sort S which is interpreted as the natural numbers with + interpreted as addition, and $x \in LV(S)$. Then

$$\overline{\mathrm{val}_g^\alpha}\Big(\big(f, (x), f(x)+1\big)\ f(0) =^S 1\Big) = \begin{cases} 1 & \text{if } \alpha(x) = 0,\ \overline{\mathrm{val}_g}(f(0)) = 0 \\ 1 & \text{if } \alpha(x) \neq 0,\ \overline{\mathrm{val}_g}(f(0)) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Let $\overline{\mathrm{val}_g}(\pi) = \{(g_1, g_2, g_3, \underline{\text{termination}}\ g_4), (g_1, g_2, g_5, \underline{\text{timeout}}\ g_6), (g_1, g_2, g_5, \underline{\text{termination}}\ g_7)\}$ and

$$\overline{\mathrm{val}_{g_i}}(F) = \begin{cases} 1 & \text{if } i \in \{3,4,5\} \\ 0 & \text{if } i \in \{1,2,6,7\} \end{cases}$$

Then

$$\overline{\mathrm{val}_g}(M(\pi)F) = \begin{cases} 0 & \text{if } M = [] \\ 1 & \text{if } M = \langle\rangle \\ 0 & \text{if } M = [[]] \\ 1 & \text{if } M = \langle\langle\rangle\rangle \\ 1 & \text{if } M = [\langle\rangle] \\ 0 & \text{if } M = \langle[]\rangle \end{cases}$$

Types of Modal Operators There are two dimensions that categorize the modal operators. The semantics and rules for operators of the same type are similar.

Definition 40. *The* types of modal operators *are given by*

- $Oper^U = \{[], [[]], [\langle\rangle]\}$
- $Oper^E = \{\langle\rangle, \langle\langle\rangle\rangle, \langle[]\rangle\}$

and orthogonally

- $Oper_U = \{[[]], \langle[]\rangle\}$
- $Oper_E = \{[\langle\rangle], \langle\langle\rangle\rangle\}$
- $Oper_F = \{[], \langle\rangle\}$

Firstly, $Oper^U$ contains those modal operators that quantify universally over traces (square brackets on the outside). And $Oper^E$ contains those modal operators that quantify existentially over traces (angular brackets on the outside).

Secondly, $Oper_U$ contains those modal operators that quantify universally over the intermediate states of a trace (square brackets on the inside). $Oper_E$ contains those modal operators that quantify existentially over the intermediate states of a trace (angular brackets on the inside). And $Oper_F$ contains those modal operators that quantify over the final states of a trace[24] (just one pair of brackets).

Duality The modal operators form three pairs (M, M') of dual operators that can be defined by each other. M' arises from M by interchanging square and angular brackets.

Definition 41. *The* dual modal operator M' *of* M *is defined by*

$$M' = \begin{cases} \langle\rangle & \text{if } M = [] \\ [] & \text{if } M = \langle\rangle \\ \langle\langle\rangle\rangle & \text{if } M = [[]] \\ [[]] & \text{if } M = \langle\langle\rangle\rangle \\ \langle[]\rangle & \text{if } M = [\langle\rangle] \\ [\langle\rangle] & \text{if } M = \langle[]\rangle \end{cases}$$

[24] There is at most one. [] quantifies universally, and ⟨⟩ quantifies existentially.

Treatment of Timeouts By the definition of $Fin(t)$ only those traces that end in a state labelled with $\underline{\text{termination}}$ are considered for the semantics of the modal operators $[]$ and $\langle\rangle$. By the definition of $IntMed(t)$ the distinction between program termination and timeout is irrelevant for the modal operators $[[]]$, $[\langle\rangle]$, $\langle[]\rangle$ and $\langle\langle\rangle\rangle$.

7.4 Update Substitution Lemma

The update substitution lemma is not an inherent property of DLTP. It applies only to first-order formulas and holds for any first order predicate logic that contains updates and conditional terms.

It states that interpreting a finite sequence U of updates and applying the interpretation to a state g (as when computing $\overline{\mathrm{val}_g}(U\ F)$) is the same as applying the induced update substitution and interpreting the substituted formula (as when computing $\overline{\mathrm{val}_g}(\sigma_U(F))$). This implies that the effect of updates can be syntactically simulated.

Lemma 2. *For any vocabulary and any structure: If U is a finite sequence of updates and F is a first-order formula, then: For all interpretation functions $\mathrm{val}_g \in \mathbf{G}$ and all assignments α.*

$$\overline{\mathrm{val}_g^\alpha}(U\ F) = \overline{\mathrm{val}_g^\alpha}(\sigma_U(F)).$$

Proof. Without loss of generality it can be assumed that no free variables of U occurs bound in F, because the renaming of the bound variables of F in the definition of σ_U does not change the semantics of F.

The proof is done by induction on the length of U.

1. $U = ()$
   $\overline{\mathrm{val}_g^\alpha}(()\ F) = \overline{\mathrm{val}_g^\alpha}(F) = \overline{\mathrm{val}_g^\alpha}(\sigma_{()}(F))$ for arbitrary val_g and α.
2. Induction basis: $U = u$ for $u = (f, (t_1, \ldots, t_n), t)$ with $\mathrm{Sign}(f) = S^1\ \ldots\ S^n\ S$ for $n \in \mathbb{N}$
   The result follows if it is proven that $\overline{\mathrm{val}_{g'}^\alpha}(r) = \overline{\mathrm{val}_g^\alpha}(\sigma_u(r))$ for terms r and $\overline{\mathrm{val}_g^\alpha}(u\ F) = \overline{\mathrm{val}_g^\alpha}(\sigma_u(F))$ for first-order formulas F for all interpretation functions val_g and all assignments α. This is proven by parallel structural induction on r and F.

   Let val_g be any interpretation function and α be any assignment and let $g' = State(g, \alpha, u)$.

- $r = x \in LV$: $\overline{\mathrm{val}_{g'}^\alpha}(r) = \overline{\mathrm{val}_{g'}^\alpha}(x) = \alpha(x) = \overline{\mathrm{val}_g^\alpha}(x) = \overline{\mathrm{val}_g^\alpha}(\sigma_u(r))$

- $r = c \in FS$, $c = f$: $\overline{\mathrm{val}_{g'}^\alpha}(r) \overset{\boxed{1}}{=} \overline{\mathrm{val}_g^\alpha}(t) \overset{\boxed{2}}{=} \overline{\mathrm{val}_g^\alpha}(\sigma_u(r))$

- $r = c \in FS$, $c \neq f$: $\overline{\mathrm{val}_{g'}^\alpha}(r) = \overline{\mathrm{val}_g^\alpha}(c) = \overline{\mathrm{val}_g^\alpha}(\sigma_u(r))$

- $r = h(r_1, \ldots, r_n)$, $h = f$:

$$\overline{\mathrm{val}_{g'}^\alpha}(r) = \mathrm{val}_{g'}(f)\big(\overline{\mathrm{val}_{g'}^\alpha}(r_1), \ldots, \overline{\mathrm{val}_{g'}^\alpha}(r_n)\big)$$

$$\overset{\boxed{3}}{=} \left\{\begin{matrix} \overline{\mathrm{val}_g^\alpha}(t) & \text{if } \overline{\mathrm{val}_{g'}^\alpha}(r_i) = \overline{\mathrm{val}_g^\alpha}(t_i) \text{ for } i \in [1;n] \\ \mathrm{val}_g(f)\big(\overline{\mathrm{val}_{g'}^\alpha}(r_1), \ldots, \overline{\mathrm{val}_{g'}^\alpha}(r_n)\big) & \text{otherwise} \end{matrix}\right\}$$

$$\overset{\boxed{4}}{=} \left\{\begin{matrix} \overline{\mathrm{val}_g^\alpha}(t) & \text{if } \overline{\mathrm{val}_g^\alpha}(\sigma_u(r_i)) = \overline{\mathrm{val}_g^\alpha}(t_i) \text{ for } i \in [1;n] \\ \mathrm{val}_g(f)\big(\overline{\mathrm{val}_g^\alpha}(\sigma_u(r_1)), \ldots, \overline{\mathrm{val}_g^\alpha}(\sigma_u(r_n))\big) & \text{otherwise} \end{matrix}\right\}$$

$$\overset{\boxed{5}}{=} \overline{\mathrm{val}_g^\alpha}\big(\big[\sigma_u(r_1) =^{S^1} t_1 \wedge \ldots \wedge \sigma_u(r_n) =^{S^n} t_n\ ?\ t\ :\ f(\sigma_u(r_1), \ldots, \sigma_u(r_n))\big]\big) \overset{\boxed{6}}{=} \overline{\mathrm{val}_g^\alpha}(\sigma_u(r))$$

- $r = h(r_1, \ldots, r_m)$, $h \neq f$:

$$\overline{\mathrm{val}_{g'}^\alpha}(r) = \mathrm{val}_{g'}(h)\big(\overline{\mathrm{val}_{g'}^\alpha}(r_1), \ldots, \overline{\mathrm{val}_{g'}^\alpha}(r_m)\big) \overset{\boxed{7}}{=} \mathrm{val}_g(h)\big(\overline{\mathrm{val}_g^\alpha}(\sigma_u(r_1)), \ldots, \overline{\mathrm{val}_g^\alpha}(\sigma_u(r_m))\big)$$

$$= \overline{\mathrm{val}_g^\alpha}\big(h(\sigma_u(r_1), \ldots, \sigma_u(r_m))\big) = \overline{\mathrm{val}_g^\alpha}(\sigma_u(r))$$

- $r = [C?r_1 : r_2]$:

$$\overline{\mathrm{val}_{g'}^\alpha}(r) \overset{\boxed{8}}{=} \left\{\begin{matrix} \overline{\mathrm{val}_{g'}^\alpha}(r_1) & \text{if } \overline{\mathrm{val}_{g'}^\alpha}(C) = 1 \\ \overline{\mathrm{val}_{g'}^\alpha}(r_2) & \text{otherwise} \end{matrix}\right\} \overset{\boxed{9}}{=} \left\{\begin{matrix} \overline{\mathrm{val}_g^\alpha}(\sigma_u(r_1)) & \text{if } \overline{\mathrm{val}_g^\alpha}(\sigma_u(C)) = 1 \\ \overline{\mathrm{val}_g^\alpha}(\sigma_u(r_2)) & \text{otherwise} \end{matrix}\right\}$$

$$\overset{\boxed{10}}{=} \overline{\mathrm{val}}_g^\alpha\big([\sigma_u(C)\ ?\ \sigma_u(r_1)\ :\ \sigma_u(r_2)]\big) \overset{\boxed{11}}{=} \overline{\mathrm{val}}_g^\alpha(\sigma_u(r))$$

- $F \in \{true, false\}$: $\overline{\mathrm{val}}_g^\alpha(u\ F) = \overline{\mathrm{val}}_g^\alpha(F) = \overline{\mathrm{val}}_g^\alpha(\sigma_u(F))$

- $F = P(r_1, \ldots, r_n)$, $P \in PS$:

  $$\overline{\mathrm{val}}_g^\alpha(u\ F) = \overline{\mathrm{val}}_{g'}^\alpha(P(r_1,\ldots,r_n)) = 1 \iff \big(\overline{\mathrm{val}}_{g'}^\alpha(r_1), \ldots, \overline{\mathrm{val}}_{g'}^\alpha(r_n)\big) \in \mathrm{val}_{g'}(P) \overset{\boxed{12}}{\iff} \big(\overline{\mathrm{val}}_g^\alpha(\sigma_u(r_1)), \ldots, \overline{\mathrm{val}}_g^\alpha(\sigma_u(r_n))\big) \in \mathrm{val}_g(P) \iff \overline{\mathrm{val}}_g^\alpha\big(P(\sigma_u(r_1), \ldots, \sigma_u(r_n))\big) = \overline{\mathrm{val}}_g^\alpha(\sigma_u(F)) = 1$$

  Therefore $\overline{\mathrm{val}}_g^\alpha\big(u\ P(r_1,\ldots,r_n)\big) = \overline{\mathrm{val}}_g^\alpha\big(\sigma_u(P(r_1,\ldots,r_n))\big)$.

- $F = \neg G$:
  $$\overline{\mathrm{val}}_g^\alpha(u\ F) = \overline{\mathrm{val}}_{g'}^\alpha(\neg G) = \neg\overline{\mathrm{val}}_{g'}^\alpha(G) \overset{\boxed{13}}{=} \neg\overline{\mathrm{val}}_g^\alpha(\sigma_u(G)) = \overline{\mathrm{val}}_g^\alpha(\neg\sigma_u(G)) = \overline{\mathrm{val}}_g^\alpha(\sigma_u(F))$$

- $F = G \diamond H$ for $\diamond \in \{\wedge, \vee\}$:
  $$\overline{\mathrm{val}}_g^\alpha(u\ F) = \overline{\mathrm{val}}_{g'}^\alpha(G \diamond H) = \overline{\mathrm{val}}_{g'}^\alpha(G) \diamond \overline{\mathrm{val}}_{g'}^\alpha(H) \overset{\boxed{14}}{=} \overline{\mathrm{val}}_g^\alpha(\sigma_u(G)) \diamond \overline{\mathrm{val}}_g^\alpha(\sigma_u(H)) = \overline{\mathrm{val}}_g^\alpha(\sigma_u(G) \diamond \sigma_u(H)) = \overline{\mathrm{val}}_g^\alpha(\sigma_u(F))$$

- $F = Qx\ G$ for $Q \in \{\forall, \exists\}$, $x \in LV(S)$, $S \in \Sigma$:

  $$\overline{\mathrm{val}}_g^\alpha(u\ F) = \overline{\mathrm{val}}_{g'}^\alpha(Qx\ G) \overset{\boxed{15}}{=} \bigodot_{v \in U(S)} \overline{\mathrm{val}}_{g'}^{\alpha_x^v}(G) \overset{\boxed{16}}{=} \bigodot_{v \in U(S)} \overline{\mathrm{val}_{State(g,\alpha_x^v,u)}^{\alpha_x^v}}(G) \overset{\boxed{17}}{=} \bigodot_{v \in U(S)} \overline{\mathrm{val}}_g^{\alpha_x^v}(u\ G)$$
  $$\overset{\boxed{18}}{=} \bigodot_{v \in U(S)} \overline{\mathrm{val}}_g^{\alpha_x^v}(\sigma_u(G)) \overset{\boxed{19}}{=} \overline{\mathrm{val}}_g^\alpha(Qx\ \sigma_u(G)) \overset{\boxed{20}}{=} \overline{\mathrm{val}}_g^\alpha(\sigma_u(F))$$

  where $\bigodot = \begin{cases} \inf & \text{if } Q = \forall \\ \sup & \text{if } Q = \exists \end{cases}$

$\boxed{1}$ by the definition of g'
$\boxed{2}$ by the definition of σ_u
$\boxed{3}$ by the definition of g'
$\boxed{4}$ by the induction hypothesis for $r_1, \ldots, r_n$
$\boxed{5}$ by the semantics of conditional terms
$\boxed{6}$ by the definition of σ_u
$\boxed{7}$ by the induction hypothesis for $r_1, \ldots, r_m$ and because u does not affect h
$\boxed{8}$ by the semantics of conditional terms
$\boxed{9}$ by the induction hypothesis for C (which is first-order), r_1 and r_2
$\boxed{10}$ by the semantics of conditional terms
$\boxed{11}$ by the definition of σ_u
$\boxed{12}$ by the induction hypothesis for $r_1, \ldots, r_n$ and because u does not affect predicate symbols
$\boxed{13}$ by the induction hypothesis for G
$\boxed{14}$ by the induction hypothesis for G and H
$\boxed{15}$ by the semantics of Q
$\boxed{16}$ because $g' = State(g, \alpha, u) = State(g, \alpha_x^v, u)$ because $x \notin FV(u)$
$\boxed{17}$ by the definition of $\overline{\mathrm{val}}_g^{\alpha_x^v}(u\ G)$
$\boxed{18}$ by the induction hypothesis for G for the state g and the assignment α_x^v
$\boxed{19}$ by the semantics of Q
$\boxed{20}$ by the definition of σ_u

---

3. Induction step: $U = u \cdot U'$

   Let val_g be any interpretation function and let α be any assignment. Then

   $$\overline{\mathrm{val}}_g^\alpha(U\ F) = \overline{\mathrm{val}}_g^\alpha(u \cdot U'\ F) \overset{\boxed{1}}{=} \overline{\mathrm{val}_{State(g,\alpha,u)}^\alpha}(U'\ F) \overset{\boxed{2}}{=} \overline{\mathrm{val}_{State(g,\alpha,u)}^\alpha}(\sigma_{U'}(F))$$

   $$\overset{\boxed{3}}{=} \overline{\mathrm{val}}_g^\alpha(u\ \sigma_{U'}(F)) \overset{\boxed{4}}{=} \overline{\mathrm{val}}_g^\alpha\big(\sigma_u(\sigma_{U'}(F))\big) \overset{\boxed{5}}{=} \overline{\mathrm{val}}_g^\alpha(\sigma_U(F))$$

$\boxed{1}$ by the definition of $\overline{\text{val}}_g^\alpha(U\ F)$

$\boxed{2}$ by the induction hypothesis for U' for the state $State(g, \alpha, u)$ and the assignment α

$\boxed{3}$ by the semantics of $u\ \sigma_{U'}(F)$

$\boxed{4}$ by the induction basis

$\boxed{5}$ by the definition of σ_U

$\square$

Example At first sight the definition of the order of update application may appear inconsistent with the update substitution lemma. In the definition of $\overline{\text{val}}_g(U\ F)$ the elements of the sequence U of updates are applied successively from left to right. If there are several updates for the same function symbol and the same argument, the later update overwrites the earlier update. In the definition of $\sigma_U(F)$ the order of applications is reversed. The following example clarifies what happens when update substitutions are applied.

Let

- $d_1, d_2 \in Terms(C)$
- q_1 and q_2 abbreviate $()^{1,I}$ and $()^{1,C}$, respectively,
- $U_1 = (nextchan, (), d_1)$, $U_2 = U_1 \cdot (cont, d_1, q_1)$ and $U = U_2 \cdot (cont, d_2, q_2)$,
- $F = cont(nextchan) =^Q q_2$,
- val_g be any interpretation function
- $\overline{\text{val}}_g(nextchan) = 1$.

Then firstly if

$$\overline{\text{val}}_g(d_1) = \overline{\text{val}}_g(d_2) = 1$$

then

$$\big(\overline{\text{val}}_g(U) * \text{val}_g(cont)\big)(i) = \begin{cases} (1, C, ()) & \text{if } i = 1 \\ \text{val}_g(cont)(i) & \text{otherwise} \end{cases}$$

and therefore $\overline{\text{val}}_g(U\ F) = 1$. And secondly if

$$\overline{\text{val}}_g(d_1) = 1 \text{ and } \overline{\text{val}}_g(d_2) = 2$$

then

$$\big(\overline{\text{val}}_g(U) * \text{val}_g(cont)\big)(i) = \begin{cases} (1, I, ()) & \text{if } i = 1 \\ (1, C, ()) & \text{if } i = 2 \\ \text{val}_g(cont)(i) & \text{otherwise} \end{cases}$$

and therefore $\overline{\text{val}}_g(U\ F) = 0$.

On the other hand

$$\sigma_U(F) = \sigma_{U_2}\big(\big[nextchan =^C d_2\ ?\ q_2\ :\ cont(nextchan)\big] =^Q q_2\big)$$

$$= \sigma_{U_1}\big(\big[nextchan =^C d_2\ ?\ q_2\ :\ [nextchan =^C d_1\ ?\ q_1\ :\ cont(nextchan)]\big] =^Q q_2\big)$$

$$= \big[d_1 =^C d_2\ ?\ q_2\ :\ [d_1 =^C d_1\ ?\ q_1 :\ cont(d_1)]\big] =^Q q_2$$

In the first case $\overline{\text{val}}_g(d_1 =^C d_2) = 1$ and therefore

$$\overline{\text{val}}_g(\sigma_U(F)) = \overline{\text{val}}_g(q_2 =^Q q_2) = 1$$

and in the second case $\overline{\mathrm{val}_g}(d_1 =^C d_2) = 0$, but $\overline{\mathrm{val}_g}(d_1 =^C d_1) = 1$ and therefore

$$\overline{\mathrm{val}_g}(\sigma_U(F)) = \overline{\mathrm{val}_g}(q_1 =^Q q_2) = 0.$$

8 The Standard Structure

Parts of the semantics of DLTP are fixed in a way that corresponds to Promela: The unwinding of composed commands and the scheduling. Other parts can differ if different structures are used. The standard structure is a structure for the standard vocabulary and gives the function and predicate symbols and the elementary commands the meaning that corresponds to the Promela semantics. After the definition several remarks and examples are given.

8.1 Definition

Definition 42. *The* standard structure

$$\mathbf{K} = (U, \mathbf{G}, O, (\mathrm{exec}^{\mathrm{A}}(c), \mathrm{eff}^{\mathrm{A}}{}_c)_{c \in \underline{\mathrm{elementary}}}, (\mathrm{exec}^{\mathrm{S}}(c,d), \mathrm{eff}^{\mathrm{S}}{}_{c,d})_{c,d \in \underline{\mathrm{elementary}}})$$

is a structure for the standard vocabulary. Its components are defined below.

Universes For $S \in \Sigma$, $U(S)$ contains a distinguished element $\perp$ that is the interpretation of undefined terms. U' is defined by $U'(S) = U(S) \setminus \{\perp\}$ for $S \in \Sigma$.

For $S \in \Sigma_E \cup \Sigma_L$ the definition of the universe function is:

$U(I) = \mathbb{Z} \cup \{\perp\}$

$U(C) = \mathbb{N} \cup \{\perp\}$

$U(Q) = \big\{(m, S, q) | m \in \mathbb{N},\ S \in \Sigma_P,\ q \in \bigcup_{i=0}^{m} U'(S)^i\big\} \cup \{\perp\}$ $\boxed{1}$

$U(P) = \{\mathrm{run}, \mathrm{fin}, \perp\}$

And for $S \in \Sigma_P \setminus \Sigma_E$ it is:
$U(S_1 \times \ldots \times S_n) = \big(U'(S_1) \times \ldots \times U'(S_n)\big) \cup \{\perp\}$

All universes are assumed to be pairwise disjoint. The above definitions are used for convenience.

States $\mathbf{G}$ is the set of all interpretation functions that satisfy the following conditions.

1. The interpretation of the predicate symbols is[25]:
   - $\mathrm{val}_g(=^S) = \big\{(a,a) \in U(S)^2 | a \in U(S)\big\}$ for $S \in \Sigma$
   - $\mathrm{val}_g(\leq^I) = \big\{(a,b) \in U'(I)^2 | a \leq b\big\} \cup \big\{(\perp,\perp)\big\}$,
     $\mathrm{val}_g(\leq^C) = \big\{(a,b) \in U'(C)^2 | a \leq b\big\} \cup \big\{(\perp,\perp)\big\}$,[26]
     $\mathrm{val}_g(\leq^S) = \big\{(a,b) \in U'(S)^2 | a\ \mathit{lex}(\mathrm{val}_g(\leq^{S_1}), \ldots, \mathrm{val}_g(\leq^{S_n}))\ b\big\} \cup \big\{(\perp,\perp)\big\}$
     for $S \in \Sigma_P \setminus \Sigma_E$ where $\mathit{lex}(\leq^1, \ldots, \leq^n)$ is the lexicographic ordering with respect to $\leq^1, \ldots, \leq^n$ $\boxed{2}$
   - $\mathrm{val}_g(\mathit{sor}^S) = \big\{(m, S', q) \in U'(Q) | S' = S\big\}$
     for $S \in \Sigma_P$ $\boxed{3}$
   - $\mathrm{val}_g(\mathit{match}^{\sigma,S}) = \big\{\big((a_1, \ldots, a_n), b_{i_1}, \ldots, b_{i_r}\big) \in U'(S) \times \prod_{i \in \sigma} U'(S_i) \mid a_i = b_i \text{ for } i \in \sigma\big\}$
     for $S \in \Sigma_P$ and $\sigma = \{i_1, \ldots, i_r\} \subseteq [1; \mathrm{Length}(S)]$ $\boxed{4}$
2. The interpretation of all rigid function symbols treats undefined arguments strictly: If $f \in FS$, $\mathrm{Sign}(f) = S^1\ \ldots\ S^n\ S$ for $n \geq 1$, then $\mathrm{val}_g(f)(a_1, \ldots, a_n) = \perp$ if $a_i = \perp$ for an $i \in [1; n]$. And the following definitions of $\mathrm{val}_g(f)$ apply only for arguments $(a_1, \ldots, a_n) \in \prod_{i=1}^{n} U'(S_i)$.

[25]The interpretation of predicate symbols with undefined arguments is 0 except for equality and comparison with two undefined arguments.

[26]The semantics of the comparison of channels is gathered from Spin ([2]). It is not specified in Promela ([1]).

3. The interpretation of the rigid function symbols is given by:

   - $\text{val}_g(c_n) = n$ for $n \in \mathbb{Z}$
   - $\text{val}_g(+)(a, b) = a + b$,
     $\text{val}_g(-)(a, b) = a - b$,
     $\text{val}_g(\cdot)(a, b) = a \cdot b$,
     $$\text{val}_g(/)(a, b) = \begin{cases} \lfloor a/b \rfloor & \text{if } b \neq 0,\ a/b \geq 0 \\ \lceil a/b \rceil & \text{if } b \neq 0,\ a/b < 0\ \boxed{5} \\ \bot & \text{if } b = 0 \end{cases}$$
   - $\text{val}_g(elem_i^S)(a_1, \ldots, a_n) = a_i$
     for $S \in \Sigma_P$ and $i \in [1; n]$ where $n = \text{Length}(S)$
   - $\text{val}_g(tuple^S)(a_1, \ldots, a_n) = (a_1, \ldots, a_n)$
     for $S \in \Sigma_P$
   - $\text{val}_g(ini^I) = 0$,
     $\text{val}_g(ini^C) = 0$,
     $\text{val}_g(ini^S) = (\text{val}_g(ini^{S_1}), \ldots, \text{val}_g(ini^{S_n}))$
     for $S \in \Sigma_P \setminus \Sigma_E$ and $n = \text{Length}(S)$
   - $\text{val}_g(\bot^S) = \bot$
     for $S \in \Sigma$
   - $\text{val}_g(nil^C) = 0$
   - $\text{val}_g(incchan)(a) = a + 1$
   - $\text{val}_g(()^{m,S}) = (m, S, ())$
     for $m \in \mathbb{N}$, $S \in \Sigma_P$
   - $\text{val}_g(cap)(m, S, q) = m$
   - $\text{val}_g(length)(m, S, q) = \text{Length}(q)$
   - $\text{val}_g(remove)((m, S, q), i) = (m, S, \text{Remove}(q, \{i\}))$
   - $$\text{val}_g(insert^S)((m, S', q), a, i) = \begin{cases} (m, S, \text{Insert}(q, a, i)) & \text{if } S = S',\ n = length(q),\ i \in [1; n+1] \\ & \qquad n < m\ \boxed{6} \\ (m, S', q) & \text{otherwise } \boxed{7} \end{cases}$$
     for $S \in \Sigma_P$
   - $$\text{val}_g(read^S)((m, S', q), i) = \begin{cases} q_i & \text{if } \overbrace{S = S'}^{\boxed{8}},\ i \in [1; \text{Length}(q)] \\ \bot & \text{otherwise} \end{cases}$$
     for $S \in \Sigma_P$
   - $$\text{val}_g(sendpos^S)((m, S', q), a) = \begin{cases} \min\ M & \text{if } S' = S \\ \bot & \text{otherwise} \end{cases}$$
     where $M = \{j \in [1; \text{Length}(q)] | (a, q_j) \in \text{val}_g(\leq^S)\} \cup \{\text{Length}(q) + 1\}$ $\boxed{9}$
   - $$\text{val}_g(recpos^{\sigma,S})((m, S', q), b_1, \ldots, b_r) = \begin{cases} \min\ M & \text{if } S' = S,\ M \neq \emptyset \\ \bot & \text{otherwise} \end{cases}$$
     where $M = \{j \in [1; \text{Length}(q)] | (q_j, b_1, \ldots, b_r) \in \text{val}_g(match^{\sigma,S})\}$ and $r = |\sigma|$ $\boxed{10}$
   - $\text{val}_g(run) = \text{run}$,
     $\text{val}_g(fin) = \text{fin}$

4. The interpretation of the non-primary non-rigid function symbols is determined by the interpretation of the primary non-rigid function symbol *cont* in the following way.

   - $\text{val}_g(lengthc)(a) = \text{Length}(q)$
     where $\text{val}_g(cont)(a) = (m, S', q)$ $\boxed{11}$
   - If $\text{val}_g(cont)(a) = (m, S', q)$ and $r = |\sigma|$:
     $$\text{val}_g(pollfirst^{\sigma,S})(a, b_1, \ldots, b_r) = \begin{cases} 1 & \text{if } \text{val}_g(recpos^{\sigma,S})((m, S', q), b_1, \ldots, b_r) = 1 \\ 0 & \text{otherwise} \end{cases}$$

and

$$\mathrm{val}_g(\mathit{pollany}^{\sigma,S})(a, b_1, \ldots, b_r) = \begin{cases} 1 & \text{if } \mathrm{val}_g(\mathit{recpos}^{\sigma,S})((m, S', q), b_1, \ldots, b_r) \neq \bot \\ 0 & \text{otherwise} \end{cases}$$

for $S \in \Sigma_P$ and $\sigma \subseteq [1; \mathrm{Length}(S)]$ $\boxed{12}$

5. The interpretations of the primary non-rigid function symbols are defined for only finitely many arguments, or formally: The sets

   $\{(p, i) \in U(I)^2 | \mathrm{val}_g(\mathit{var}^S)(p, i) \neq \bot\}$ for $S \in \Sigma_P$

   $\{i \in U(C) | \mathrm{val}_g(\mathit{cont})(i) \neq \bot\}$

   and

   $\{i \in U(I) | \mathrm{val}_g(\mathit{proc})(i) \neq \bot\}$

   are finite.

That implies that a state g is uniquely characterized by the restriction of the interpretation function val_g to the primary non-rigid function symbols. And all interpretations of these symbols that are defined for only finitely many arguments induce a state.

$\boxed{1}$ The three components of a queue are its capacity, its sort and the actual content. Only objects of the matching sort may be stored in a queue. The content of a queue of capacity 0 is always the empty sequence; such queues are used for synchronous transfers.

$\boxed{2}$ For elementary sorts the ordering is the usual ordering for numbers where $\bot$ is only comparable to itself. For composed sorts the lexicographic ordering is used. All orderings are linear if restricted to defined elements. For channels this definition means that channels that have been created earlier are smaller.

$\boxed{3}$ sor^S is true if its argument is a queue of the sort S.

$\boxed{4}$ $\mathit{match}^{\sigma,S}$ is true if its first argument agrees in the components indexed by σ with the remaining arguments. This is the pattern matching condition for random receive commands.

$\boxed{5}$ If the result of a division is not a whole number, it is rounded to zero. Division by zero returns $\bot$.[27]

$\boxed{6}$ The sorts of the queue and the inserted element must match, and the capacity may not be exceeded.

$\boxed{7}$ If an insertion is not possible due to the sort or the capacity of the queue the argument is returned unchanged.

$\boxed{8}$ The return value of read^S must be of the sort S. This is only possible if the queue has the same sort. Otherwise read^S returns the undefined element.

$\boxed{9}$ $\mathit{sendpos}^S$ maps a queue and an object a that is to be inserted into the queue to the position of insertion for a sorted <u>send</u> command. This position is the lowest possible position such that a is inserted in front of a bigger object, or the length of the queue plus one if no such position exists. If only sorted sending is used to insert elements into a queue, it is guaranteed that the elements in the queue are sorted in ascending order.
If the sorts of the queue and the element that is to be inserted do not match, it returns the undefined element.

$\boxed{10}$ $\mathit{recpos}^{\sigma,S}$ maps a queue and a pattern that is to be used in a random <u>receive</u> command from that queue to the position of the object that is retrieved. This position is the lowest possible position such that the pattern is matched. The condition $S = S'$ is not really needed because it is implied by $M \neq \emptyset$.
If no element matches it returns the undefined element.
This and the preceding definition reveal a (surprising) symmetry between sorted sending and random receiving.

$\boxed{11}$ The program function symbol *lengthc* is reduced to the logical function symbol *length*. *lengthc* can be used inside programs to obtain the length of the queue a channel points to.

$\boxed{12}$ The program function symbols $\mathit{pollfirst}^{\sigma,S}$ and $\mathit{pollany}^{\sigma,S}$ are reduced to the logical function symbol $\mathit{recpos}^{\sigma,S}$. These symbols can be used inside programs to test the executability of the corresponding <u>receive</u> command. Since Promela only uses integer and no Boolean data types 1 is used for truth and 0 for falsity.

The above definitions determine the semantics of all terms and first-order formulas. Therefore the extension of val_g to terms can be used in the definition of the remaining components of the standard structure.

[27] The precise interpretation of division is not specified in [1]. The rounding to zero is gathered from the behavior of Spin ([2]). Division by zero, however, leads to a crash of Spin ([2]).

Examples Most interpretations are intuitively clear, but in the other cases some examples are helpful. Let $g \in \mathbf{G}$, $S = I \times I$ and

$$q = \mathrm{val}_g(\mathit{cont})(1) = \Big(5, S, \big((1,1),(1,3),(2,3),(2,1)\big)\Big).$$

Then

$$q \in \mathrm{val}_g(\mathit{sor}^S)$$

$$\big((2,1,3),2,3\big) \in \mathrm{val}_g(\mathit{match}^{\{1,3\},I\times I\times I})$$

$$\big((3,1,3),2,3\big) \notin \mathrm{val}_g(\mathit{match}^{\{1,3\},I\times I\times I})$$

$$\mathrm{val}_g(\mathit{insert}^S)(q,(1,3),4) = \Big(5, S, \big((1,1),(1,3),(2,3),(1,3),(2,1)\big)\Big)$$

$$\mathrm{val}_g(\mathit{remove}^S)(q,5) = q$$

$$\mathrm{val}_g(\mathit{read}^{I\times C})(q,3) = \bot$$

$$\mathrm{val}_g(\mathit{sendpos}^S)(q,(1,3)) = 2 \text{ where } M = \{2,3,4,5\}$$

$$\mathrm{val}_g(\mathit{recpos}^{\{2\},S})(q,3) = 2 \text{ where } M = \{2,3\}$$

$$\mathrm{val}_g(\mathit{recpos}^{\{1,2\},S})(q,1,2) = \bot \text{ where } M = \emptyset$$

$$\mathrm{val}_g(\mathit{lengthc})(1) = 4$$

$$\mathrm{val}_g(\mathit{pollfirst}^{\{2\},S})(1,3) = 0$$

$$\mathrm{val}_g(\mathit{pollany}^{\{2\},S})(1,3) = 1$$

Origin The interpretation of the primary non-rigid function symbols in the origin O is given by

$$\mathrm{val}_O(f) = \begin{cases} u \mapsto \bot & \text{if } f = \mathit{cont} \text{ or } f = \mathit{proc} \\ (u,v) \mapsto \bot & \text{if } f = \mathit{var}^S,\ S \in \Sigma_P \\ -1 & \text{if } f = \mathit{nextpid} \\ 1 & \text{if } f = \mathit{nextchan} \end{cases}$$

Informally the origin is the state in which no objects exist (queues, processes and program variables are undefined); the first instantiated process receives the process ID -1 and the first used channel is 1.

The Initialization Algorithm In Promela the initial value of a channel variable is actually the initial value of the queue it points to.[28] Therefore $\mathit{Ini}(\mathrm{PT})(C,i)$ cannot simply be a term of the sort channel, and the very general definition of $\mathit{Ini}(\mathrm{PT})$ (Definition 11) is used, which allows in particular that $\mathit{Ini}(\mathrm{PT})(C,i)$ is an empty queue. Therefore the initialization algorithm is necessary.

The initialization algorithm computes the modified initialization mapping $\mathit{MIni}(\mathrm{PT})$. For $\mathrm{PT} \in \mathbf{P}$, $S \in \Sigma_P$ and $i \in [1; \mathit{NmbVar}(\mathrm{PT},S)]$, $\mathit{MIni}(\mathrm{PT})(S,i)$ is a program term of the sort S and is used to initialize the local program variable $\mathit{var}^S(c_p, c_i)$ if a new process with ID p and of the type PT is instantiated.

The construction of new queues means that the initialization of program variables has side effects. The details of these side effects are also computed by the algorithm. The output of the algorithm is used in the definition of the effect of a <u>run</u> command.

```
1  INPUT: $\mathrm{PT} \in \mathbf{P}$
2  $n := 1$
3  $C_n := \mathit{nextchan}$
4  FOR ALL $S \in \Sigma_P$ such that $V_S := \mathit{NmbVar}(\mathrm{PT},S) \neq 0$
5          IF $S \neq C$ THEN
6              FOR $i := 1$ TO $V_S$
7                  IF $\mathit{Ini}(\mathrm{PT})(S,i) \in \mathit{PTerms}(S)$
8                      $\mathit{MIni}(\mathrm{PT})(S,i) := \mathit{Ini}(\mathrm{PT})(S,i)$
                   ELSE
```

[28] See section 9.3.3 for a detailed discussion.

```
$\boxed{9}$                     $MIni(\mathrm{PT})(S,i) := ini^S$
                       NEXT $i$
$\boxed{10}$               ELSE
                       FOR $i := 1$ TO $V_C$
$\boxed{11}$                     IF $Ini(\mathrm{PT})(S,i) = ()^{m,S'}$ for some $m \in \mathbb{N}$, $S' \in \Sigma_P$
$\boxed{12}$                        $MIni(\mathrm{PT})(S,i) := C_n$
$\boxed{13}$                        $Q_n := Ini(\mathrm{PT})(S,i)$
$\boxed{14}$                        $n := n+1$
$\boxed{15}$                        $C_n := incchan(C_{n-1})$
                          ELSE
$\boxed{16}$                        $MIni(\mathrm{PT})(S,i) := ini^C$
                       NEXT $i$
     NEXT $S$
$\boxed{17}$ OUTPUT: $MIni(\mathrm{PT})$, $n$, $(C_i, Q_i)$ for $i \in [1;n]$, $C_{n+1}$
```

$\boxed{1}$ PT is the type of the new process.

$\boxed{2}$ n counts the number of new queues that are constructed.

$\boxed{3}$ C_1 is a term that gives the channel for the first new queue. Its value is given by the state in which the process is instantiated.

$\boxed{4}$ All program sorts are treated independently from each other. The finiteness condition on $NmbVar$ ensures the termination of the algorithm.

$\boxed{5}$ All non-channel sorts are treated in the same way: ...

$\boxed{6}$ ... All local variables are initialized, ...

$\boxed{7}$ ... depending on whether the initialization specified by $Ini(\mathrm{PT})$ makes sense[29], ...

$\boxed{8}$ ... with the specified initialization, ...

$\boxed{9}$ ... or with the standard initialization.

$\boxed{10}$ The local variables of the sort channel are treated in a special way.

$\boxed{11}$ If the initialization specified by $Ini(\mathrm{PT})$ is an empty queue, ...

$\boxed{12}$ ... the variable is initialized with the next channel value, ...

$\boxed{13}$ ... the initial value of the new queue is stored in Q_n, ...

$\boxed{14}$ ... n is incremented to count the number of new queues, ...

$\boxed{15}$ ... and C_n is a term that gives the channel for the next new queue.

$\boxed{16}$ In all other cases the variable is initialized with the standard initialization (which is usually the pointer to the undefined queue).

$\boxed{17}$ $MIni(\mathrm{PT})$ is the modified initialization mapping, (C_i, Q_i) for $i \in [1;n]$ give the new queues that are constructed, and C_{n+1} gives the new value of $nextchan$.

Asynchronous Executability Conditions and Effect Functions For $c \in$ <u>elementary</u> the executability conditions $\mathrm{exec}^{\mathrm{A}}(c)$ and the effect functions $\mathrm{eff^A}_c$ are as below:

- If c is of the type <u>assignment</u> and $c = var^S(t_1, t_2) := t$:

  $\mathrm{exec}^{\mathrm{A}}(c) = \mathbf{G}$ $\boxed{1}$

  $\mathrm{val}_{\mathrm{eff^A}_c(g)} = (var^S, (\overline{\mathrm{val}_g}(t_1), \overline{\mathrm{val}_g}(t_2)), \overline{\mathrm{val}_g}(t)) * \mathrm{val}_g$ $\boxed{2}$

- If c is of the type <u>expression</u> and $c = t$ for $t \in Terms(S)$:

  $\mathrm{exec}^{\mathrm{A}}(c) = \{g \in \mathbf{G} | \overline{\mathrm{val}_g}(t) \neq \overline{\mathrm{val}_g}(ini^S)\}$ $\boxed{3}$

  $\mathrm{eff^A}_c(g) = g$ $\boxed{4}$

- If c is of the type <u>break</u> or <u>else</u> $\boxed{5}$:

  $\mathrm{exec}^{\mathrm{A}}(c) = \mathbf{G}$

  $\mathrm{eff^A}_c(g) = g$

[29]The set of rigid program terms of the sort C is $\{ini^C\}$ which entails that components of the sort C of a program variable of a composed sort can only be initialized with a trivial value (but that is still more than Promela allows).

- If c is of the type <u>run</u> and
  - $Proc(c) = \mathrm{PT}$ and $Par(c) = p$,
  - $V_S = NmbVar(\mathrm{PT}, S)$ and $P_S = NmbPar(\mathrm{PT}, S)$ for $S \in \Sigma_P$,
  - the following objects are computed by the initialization algorithm:
    * the modified initialization mapping $MIni(\mathrm{PT})$,
    * the number of new queues n,
    * for each $i \in [1; n]$ the channel C_i and the queue Q_i for the new queues,
    * the new next free channel C_{n+1},

  then:

  $$\mathrm{exec}^{\mathrm{A}}(c) = \mathbf{G}\ \boxed{6}$$

  $$\mathrm{val}_{\mathrm{eff}^{\mathrm{A}}{}_{c}(g)} = U * \mathrm{val}_g$$

  where U consists of the following interpreted updates (The order is irrelevant except for the last one, which must be at the last position of U.):

  - $(proc, \mathrm{val}_g(nextpid), \mathrm{run})\ \boxed{7}$
  - $\Big(var^S, \big(\mathrm{val}_g(nextpid), i\big), \overline{\mathrm{val}_g}\big(MIni(PT)(S,i)\big)\Big)$ for $S \in \Sigma_P$, $i \in [1; V_S]$ $\boxed{8}$
  - $(cont, \overline{\mathrm{val}_g}(C_i), \overline{\mathrm{val}_g}(Q_i))$ for $i \in [1; n]$ $\boxed{9}$
  - $(nextchan, (), \overline{\mathrm{val}_g}(C_{n+1}))\ \boxed{10}$
  - $\Big(var^S, \big(\mathrm{val}_g(nextpid), i\big), \overline{\mathrm{val}_g}(p(S, i - V_S))\Big)$ for $S \in \Sigma_P$, $i \in [V_S + 1; V_S + P_S]$ $\boxed{11}$
  - $(nextpid, (), \mathrm{val}_g(nextpid) + 1)\ \boxed{12}$

- If c is of the type <u>send</u> and $\overline{\mathrm{val}_g}(cont(Chan(c))) = \bot$, then $\mathrm{exec}^{\mathrm{A}}(c) = \emptyset$.
  If $\overline{\mathrm{val}_g}(cont(Chan(c))) = (m, S', q)$, let $Sor(c) = S$ and $\overline{\mathrm{val}_g}(Arg(c)) = a$:

  $$g \in \mathrm{exec}^{\mathrm{A}}(c) \Leftrightarrow \mathrm{Length}(q) < m \text{ and } S' = S\ \boxed{13}$$

  The effect depends on the type of the <u>send</u> command:

  - $Type(c) =!$ (normal send): $\mathrm{val}_{\mathrm{eff}^{\mathrm{A}}{}_{c}(g)} = \big(cont, \mathrm{val}_g(Chan(c)), (m, S', \overbrace{q \cdot (a)}^{\boxed{14}})\big) * \mathrm{val}_g$
  - $Type(c) =!!$ (sorted send): $\mathrm{val}_{\mathrm{eff}^{\mathrm{A}}{}_{c}(g)} = \big(cont, \mathrm{val}_g(Chan(c)), (m, S', \overbrace{\mathrm{Insert}(q, a, p)}^{\boxed{15}})\big) * \mathrm{val}_g$
    where $p = \mathrm{val}_g(sendpos^S)((m, S', q), a)$

- If c is of the type <u>receive</u> with $Chan(c) = C$, $Sor(c) = S$, $\mathrm{Length}(S) = n$, $Pos(c) = \sigma = \{i_1, \ldots, i_r\}$ and $Arg(c) = (a_i)_{i\in\sigma}$, the four types of <u>receive</u> commands must be distinguished:

$Type(c)$	normal receive	random receive
remove	?	??
poll	? <	?? <

Normal receive commands retrieve the first element of the queue if it matches. Random receive commands retrieve the first matching element of the queue if there is any. Remove commands remove the retrieved element from the queue if there is one, poll commands do not.

If $p = \overline{\mathrm{val}_g}\big(recpos^{\sigma,S}(cont(C), a_{i_1}, \ldots, a_{i_r})\big)$ gives the position of the first matching element of the queue, then:

$$g \in \mathrm{exec}^{\mathrm{A}}(c) \Leftrightarrow \begin{cases} p = 1 & \text{if } Type(c) \in \{?, ?<\}\ \boxed{16} \\ p \neq \bot & \text{if } Type(c) \in \{??, ??<\}\ \boxed{17} \end{cases}$$

If c is executable, let $\overline{\mathrm{val}_g}(cont(C)) = (m, S', q)$ and let $b = q_p$ be the first matching element in q. Normal and random receive commands have the same effect relative to p: They read the element at position p onto the variables. The remove commands additionally remove this element from the queue:

$$\mathrm{val}_{\mathrm{eff}^{\mathrm{A}}{}_{c}(g)} = U * \mathrm{val}_g$$

where U consists of the following interpreted updates (The order is the ascending order of i, but it is irrelevant if all program variables in $Var(c)$ are different.):

- $(var^{S_i}, (\overline{\mathrm{val}_g}(s_i), \overline{\mathrm{val}_g}(t_i)), b_i)$ with $Var(c)_i = var^{S_i}(s_i, t_i)$ for $i \in [1; n] \setminus \sigma$ $\boxed{18}$
- $\big(cont, \overline{\mathrm{val}_g}(C), (m, S', \mathrm{Remove}(q, \{p\}))\big)$ if $Type(c) \in \{?, ??\}$ $\boxed{19}$

- If c is of the type <u>end</u>, $c = \mathrm{END}(\mathrm{PT}, t)$ and $p = \overline{\mathrm{val}_g}(t)$:

  $\mathrm{exec}^{\mathrm{A}}(c) = \mathbf{G}$ $\boxed{20}$

  $\mathrm{val}_{\mathrm{eff}^{\mathrm{A}}{}_c(g)} = U * \mathrm{val}_g$

  where U consists of the following interpreted updates (The order is irrelevant.):

  - $(proc, p, \mathrm{fin})$ $\boxed{21}$
  - $(var^S, (p, i), \perp)$ for $S \in \Sigma_P$ and $i \in [1; NmbVar(\mathrm{PT}, S) + NmbPar(\mathrm{PT}, S)]$ $\boxed{22}$

Synchronous Executability Conditions and Effect Functions

$$\mathrm{exec}^{\mathrm{S}}(c, d) = \begin{cases} G(c,d) & \text{if } c \in \underline{\text{send}},\ d \in \underline{\text{receive}} \\ \emptyset & \text{otherwise} \end{cases} \boxed{23}$$

where

$$G(c,d) = \Big\{ g \in \mathbf{G} \,\Big|\, \overbrace{\overline{\mathrm{val}_g}(Chan(c)) = \overline{\mathrm{val}_g}(Chan(d)),\ \overline{\mathrm{val}_g}\big(cont(Chan(c))\big) = (0, S, ()),\ S = S'}^{\boxed{24}},$$
$$\overbrace{\overline{\mathrm{val}_g}(match^{\sigma,S}(Arg(c), a_{i_1}, \ldots, a_{i_r})) = 1}^{\boxed{25}} \Big\}$$

with $S = Sor(c)$, $S' = Sor(d)$, $a = Arg(d)$ and $\sigma = Pos(d) = \{i_1, \ldots, i_r\}$

$\mathrm{eff}^{\mathrm{S}}{}_{c,d}(g)$ consists of the following updates[30] (The order is the ascending order of i, but it is irrelevant if all program variables in $Var(c)$ are different.):

$\big(var^{S_i}, (\overline{\mathrm{val}_g}(s_i), \overline{\mathrm{val}_g}(t_i)), \overline{\mathrm{val}_g}(Arg(c)_i)\big)$ with $Var(d)_i = var^{S_i}(s_i, t_i)$ for $i \in [1; n] \setminus \sigma$ $\boxed{26}$

$\boxed{1}$ Assignments are always executable.

$\boxed{2}$ The effect of an assignment is the appropriate change of a program variable.

$\boxed{3}$ Expressions are executable if they are non-zero. For arbitrary sorts the initial value determines what zero means.

$\boxed{4}$ Expressions have no effect.

$\boxed{5}$ BREAK and ELSE are always executable and have no effect. Their special semantics is defined by the unwinding and remaining program functions.

$\boxed{6}$ Starting a new process is always executable.[31]

$\boxed{7}$ The process state is updated.

$\boxed{8}$ The program variables that are not parameters are initialized with the values computed by the initialization algorithm.

$\boxed{9}$ The new queues are constructed as computed by the initialization algorithm.

$\boxed{10}$ *nextchan* is set to the next channel value as computed by the initialization algorithm.

$\boxed{11}$ The parameters are initialized with the values received from the instantiating process.

$\boxed{12}$ *nextpid* is incremented to give the next free process ID.

$\boxed{13}$ Normal and sorted send commands are executable if there is a free slot in the queue and if the sort of the sent element is the same as the sort of the queue the channel points to.

$\boxed{14}$ Normal send commands add the sent element to the end of the queue.

$\boxed{15}$ Sorted send commands insert the sent element at the position for sorted sending, which is given by p.

$\boxed{16}$ Normal receive commands are executable if the first element of the queue matches the pattern of the command.

$\boxed{17}$ Random receive commands are executable if any element of the queue matches the pattern of the command. In both cases if $g \in \mathrm{exec}^{\mathrm{A}}(c)$, $S = S'$ holds automatically by the semantics of $recpos^{\sigma,S}$.

$\boxed{18}$ The variables of the command are updated with the values received from the channel.

$\boxed{19}$ The received element is removed from the queue if c is a remove command.

$\boxed{20}$ Terminating a process is always executable.

[30] $\mathrm{eff}^{\mathrm{S}}{}_{c,d}$ only needs to be defined if $\mathrm{exec}^{\mathrm{S}}(c,d) \neq \emptyset$.

[31] This is the main difference to Promela.

$\boxed{21}$ The state of the terminated process is set to fin.

$\boxed{22}$ All local variables of the terminated process are set to $\perp$.

$\boxed{23}$ Only a pair of a $\underline{\text{send}}$ and a $\underline{\text{receive}}$ command can be synchronously executable. $G(c, d)$ contains the states in which a synchronous transfer is possible.

$\boxed{24}$ This requires that $Chan(c)$ and $Chan(d)$ point to the same queue and that c and d have the same sort as this queue and that the queue is synchronous, i. e. has capacity 0.

$\boxed{25}$ This is the same pattern matching condition as for asynchronous $\underline{\text{receive}}$ commands.

$\boxed{26}$ The sent values are assigned to the variables.

One thing remains to be proven:

Lemma 3. *The standard structure satisfies the reachability condition (and therefore is indeed a structure). In fact all states of the standard structure are reachable from the origin.*

Proof. **G** contains all possible interpretations of the primary non-rigid functions symbols that are defined for only finitely many arguments. The origin satisfies this property; and this property is preserved under the application of an effect or an interpreted update. Therefore all reachable states are elements of **G**. And any state in **G** can be reached from the origin by applying finitely many interpreted updates. □

8.2 Remarks

Equivalence to the Promela Semantics Although the Promela language reference in [1] is much closer to a formal specification of the semantics of a programming language than the references for most other programming languages it still falls short.

Therefore it is impossible to formally prove that the semantics of programs in DLTP is the same as of their counterparts in Promela. That is rather a meta-theorem that can only be postulated. In section 9 the equivalence is informally discussed.

Undefined Values A program variable can be assigned the undefined value if a command accesses a global program variable that has not been initialized, reads from a queue that holds an undefined value or divides by zero. Therefore programs can use undefined values in all situations although $\perp^S$ is not a program function symbol for $S \in \Sigma_P$.

This is not usual in programming languages and not part of Promela. Although this makes DLTP more expressive than other languages, it may be seen as a disadvantage. However, it would be very difficult to avoid this effect. There are essentially two possibilities: Making commands that compute undefined values not executable or replacing $\perp \in U(S)$ with $\overline{\text{val}_O}(ini^S)$ for $S \in \Sigma_P$ when interpreting program terms. Both have big disadvantages of their own.

Therefore the definition of DLTP does not address this problem. It is deemed the task of the programmer to take care that the programs do not introduce undefined values into the computation or that they handle them appropriately. DLTP supports this with the recommendation of sensible program execution (see below).

Sensible Program Execution The definitions of DLTP are very general and do not exclude lots of senseless cases. In particular they do not enforce conditions on states that relate the interpretations of the primary non-rigid function symbols to each other.[32] The intuitive conditions that they should always satisfy are:

- $proc$ is defined precisely for the arguments between -1 and $\overline{\text{val}_g}(nextpid) - 1$. Then the value of $nextpid$ is the next free process ID.
- For $i \in [1; \text{Length}(\pi)]$ if $\text{Last}(\pi_i) \in \underline{\text{end}}$ the arguments of the $\underline{\text{end}}$ command give the type and the ID of π_i.
- If π_i is a process of type PT and with ID $n \in \mathbb{Z}$, then $\overline{\text{val}_g}(proc(c_n)) = \text{run}$, and for $S \in \Sigma_P$ precisely those local program variables $var^S(c_n, c_i)$ are defined for which $i \in [1; NmbVar(\text{PT}, S) + NmbPar(\text{PT}, S)]$.

[32]It would not be reasonable to do that because it is always possible to give updates that change into a forbidden state, and to impose further restrictions on updates is extremely awkward.

- For $n \in \mathbb{Z}$ if $\overline{\text{val}}_g(proc(c_n)) \neq$ run then all program variables $var^S(c_n, c_i)$ for $S \in \Sigma_P$ and $i \in \mathbb{Z}$ are undefined.
- $cont$ is defined precisely for the arguments between 1 and $\overline{\text{val}}_g(nextchan) - 1$. Then the value of $nextchan$ is the next free channel, and the channel value 0 always points to the undefined queue, which means that nil^C is indeed the null pointer.

Depending on the programs and the states in which they are executed it is possible that these conditions are arbitrarily violated. It is deemed the task of the user to exclude this. But DLTP does guarantee that these conditions hold if the initial state and program are chosen in the following sensible way:

- The execution starts in the origin of the standard structure.
- The initial program consists only of the command RUN GLOBAL.
- $Body$(GLOBAL) = RUN init, which means that a special process of the type GLOBAL receives the ID -1, declares the global program variables, instantiates the first real process and terminates without the execution of a command of the type $\underline{\text{end}}$.
- All process types PT $\in decl$(init) satisfy Last($Body$(PT)) $\in \underline{\text{end}}$.
- For $S \in \Sigma_P$ only those global program variables $var^S(glob, c_i)$ occur in the bodies of all process types in $decl$(GLOBAL) for which $i \in [1; NmbVar(\text{GLOBAL}, S)]$.
- No division by zero is executed.[33]

The above is sufficient to describe the execution of Promela programs, because those do not depend on the initial state.[34] The example program of section 4.4 is constructed in that way. But there are other situations, in which the execution does not start in the origin, for example when

1. $\overline{\text{val}}_O([[\pi]][\pi']F)$ is computed or
2. $P \rightarrow [\pi']Q$ is proven to hold in all states.

In the first case π' is executed from every intermediate state of π. The processes, queues and program variables that are defined in these states depend on π and may vary almost arbitrarily. Furthermore the processes in π' can contain any program variable. Therefore it is reasonable to require π' to consist of only one process of the form

RUN init2 Par

where Par only accesses global program variables. If the execution of π' starts in g, then $nextpid$ and $nextchan$ have sensible values, init2 receives the ID $\overline{\text{val}}_g(nextpid)$, and π' can be executed sensibly with two exceptions: Firstly, all processes that were executed in g and their local program variables stay defined during the execution of π' (but are inaccessible for init2) although these processes are not executed anymore (see the example in section 8.3.2). And secondly, for the programmer of π' it is not clear which global program variables have been defined by π.

In the second case π' should also consist of only one process of the form

RUN PT Par

where Par only accesses global program variables. P should restrict the initial state in a way that excludes all senseless possibilities: program variables should be undefined[35] except for the global program variables, and $nextpid$ and $nextchan$ should have sensible values (see the example in section 8.3.2). If PT is meant to declare the global program variables, P must imply $nextpid =^I glob$.

In both cases π' can still access all global program variables and all queues that are pointed to by such global program variables (Such a queue may contain the pointer to another queue.), which is sufficient to conveniently use programs that depend on the initial state.[36]

[33] This can be enforced by using conditional terms or atomic sequences.
[34] The initialization of the global program variables is part of the execution, not a property of the initial state.
[35] This is possible for only finitely many sorts, because for each sort there is a different function symbol.
[36] Programs that depend on the local program variables of some process can still be written, but they will rarely be necessary. And in that case it is more elegant to assign the needed information to global variables anyway.

8.3 Examples

8.3.1 Continued Example

The semantics of the example program, denoted by π, and the example formulas of section 4.4 can now be determined in the standard structure. The semantics is computed in the origin $g_0 = O$. At each step only one command is executable. Therefore $\overline{\mathrm{val}_{g_0}}(\pi)$ contains only one trace. For $i \in [0;11]$

$$\overline{\mathrm{val}_{g_i}}(\pi_i) = \{(g_i, \ldots, \underline{\mathrm{termination}}\ g_{11})\}$$

and for $i \in [0;10]$

$$\mathrm{val}_{g_{i+1}} = U_i * \mathrm{val}_{g_i}$$

where U_i and π_i are given below. The underlined command of the program π_i is executed in the state g_i giving the state g_{i+1} via the interpreted update U_i (Since all local variables are of the sort C, the superscript C of local variables is left out.):

- $\pi_0 = \underline{\mathrm{RUN\ GLOBAL}}$

  The initialization algorithm returns $MIni(\mathrm{GLOBAL})(I,1) = c_{41}$, $n = 0$ and $C_1 = nextchan$. Then:

  $U_0 = (proc, -1, \mathrm{run}) \cdot (var^I, (-1,1), 41) \cdot (nextchan, (), 1) \cdot (nextpid, (), 0)$

- $\pi_1 = \underline{\mathrm{RUN\ init}}$

  The initialization algorithm returns $MIni(\mathrm{init})(C,1) = nextchan$, $MIni(\mathrm{init})(C,2) = incchan(nextchan)$, $n = 2$, $(C_1, Q_1) = (nextchan, ()^{1,C})$, $(C_2, Q_2) = (incchan(nextchan), ()^{1,I})$ and $C_3 = incchan(incchan(nextchan))$. Then:

  $U_1 = (proc, 0, \mathrm{run}) \cdot (var^C, (0,1), 1) \cdot (var^C, (0,2), 2) \cdot \big(cont, 1, (1, C, ())\big) \cdot \big(cont, 2, (1, I, ())\big) \cdot (nextchan, (), 3) \cdot (nextpid, (), 1)$

- $\pi_2 = \underline{\mathrm{RUN\ A}\ p_1};\ \mathrm{RUN\ B}\ p_2;\ X_1\ !\ X_2;\ \mathrm{END}(\mathrm{init}, nextpid - c_1)$

  The initialization algorithm returns $MIni(\mathrm{A})(C,1) = ini^C$, $n = 0$ and $C_1 = nextchan$. p_1 is given by $p_1(C,1) = var^C(nextpid - c_1, c_1)$. Then:

  $U_2 = (proc, 1, \mathrm{run}) \cdot (var^C, (1,1), 0) \cdot (nextchan, (), 3) \cdot (var^C, (1,2), 1) \cdot (nextpid, (), 2)$

- $\pi_3 = \underline{\mathrm{RUN\ B}\ p_2};\ X_1\ !\ X_2;\ \mathrm{END}(\mathrm{init}, nextpid - c_1 - c_1)\ |$
  $\quad X_2\ ?\ \emptyset\ (X_1)\ ();\ X_1\ !\ c_{42};\ \mathrm{END}(\mathrm{A}, nextpid - c_1)$

  The initialization algorithm returns $n = 0$ and $C_1 = nextchan$. p_2 is given by $p_2(C,1) = var^C(nextpid - c_1 - c_1, c_2)$. Then:

  $U_3 = (proc, 2, \mathrm{run}) \cdot (nextchan, (), 3) \cdot (var^C, (2,1), 2) \cdot (nextpid, (), 3)$

- $\pi_4 = \underline{X_1\ !\ X_2};\ \mathrm{END}(\mathrm{init}, nextpid - c_1 - c_1 - c_1)\ |$
  $\quad X_2\ ?\ \emptyset\ (X_1)\ ();\ X_1\ !\ c_{42};\ \mathrm{END}(\mathrm{A}, nextpid - c_1 - c_1)\ |$
  $\quad X_1\ ?\ \emptyset\ (Y_1^I)\ ();\ \mathrm{END}(\mathrm{B}, nextpid - c_1)$

  $\overline{\mathrm{val}_{g_4}}\big(var^C(nextpid - c_1 - c_1 - c_1, c_1)\big) = 1$ and $\overline{\mathrm{val}_{g_4}}\big(var^C(nextpid - c_1 - c_1 - c_1, c_2)\big) = 2$. Then:

  $U_4 = (cont, 1, (1, C, (2))$

- $\pi_5 = \underline{\mathrm{END}(\mathrm{init}, nextpid - c_1 - c_1 - c_1)}\ |$
  $\quad X_2\ ?\ \emptyset\ (X_1)\ ();\ X_1\ !\ c_{42};\ \mathrm{END}(\mathrm{A}, nextpid - c_1 - c_1)\ |$
  $\quad X_1\ ?\ \emptyset\ (Y_1^I)\ ();\ \mathrm{END}(\mathrm{B}, nextpid - c_1)$

  $U_5 = (proc, 0, \mathrm{fin}) \cdot (var^C, (0,1), \bot) \cdot (var^C, (0,2), \bot)$

- $\pi_6 = \underline{X_2\ ?\ \emptyset\ (X_1)\ ()};\ X_1\ !\ c_{42};\ \mathrm{END}(\mathrm{A}, nextpid - c_1 - c_1)\ |$
  $\quad \underline{X_1\ ?\ \emptyset\ (Y_1^I)\ ()};\ \mathrm{END}(\mathrm{B}, nextpid - c_1)$

  $\overline{\mathrm{val}_{g_6}}\big(var^C(nextpid - c_1 - c_1, c_2)\big) = 1$ and $\mathrm{val}_{g_6}(cont)(1) = (1, C, (2))$. Then:

  $U_6 = (var^C, (1,1), 2) \cdot \big(cont, 1, (1, C, ())\big)$

- $\pi_7 = \underline{X_1\ !\ c_{42}};\ \mathrm{END}(\mathrm{A}, nextpid - c_1 - c_1)\ |$
  $\quad X_1\ ?\ \emptyset\ (Y_1^I)\ ();\ \mathrm{END}(\mathrm{B}, nextpid - c_1)$

  $\overline{\mathrm{val}_{g_7}}\big(var^C(nextpid - c_1 - c_1, c_1)\big) = 2$. Then:

  $U_7 = (cont, 2, (1, I, (42)))$

- $\pi_8 = \underline{\mathrm{END}(\mathrm{A}, nextpid - c_1 - c_1)|}$
  $\qquad X_1 \ ?\ \emptyset\ (Y_1^I)\ ();\ \mathrm{END}(\mathrm{B}, nextpid - c_1)$
  $U_8 = (proc, 1, \mathrm{fin}) \cdot (var^C, (1,1), \bot) \cdot (var^C, (1,2), \bot)$
- $\pi_9 = \underline{X_1 \ ?\ \emptyset\ (Y_1^I)\ ()};\ \mathrm{END}(\mathrm{B}, nextpid - c_1)$
  $\overline{\mathrm{val}}_{g_9}\big(var^C(nextpid - c_1, c_1)\big) = 2$ and $\mathrm{val}_{g_9}(cont)(2) = (1, I, (42))$. Then:
  $U_9 = (var^I, (-1,1), 42) \cdot \big(cont, 2, (1, I, ())\big)$
- $\pi_{10} = \underline{\mathrm{END}(\mathrm{B}, nextpid - c_1)}$
  $U_{10} = (proc, 2, \mathrm{fin}) \cdot (var^C, (2,1), \bot)$
- $\pi_{11} = \epsilon$

The execution stops in the final state g_{11} in which the interpretation of the primary non-rigid function symbols is given by

$$\mathrm{val}_{g_{11}}(f) = \begin{cases} (p,i) \mapsto \left\{\begin{matrix} 42 & \text{if } p = -1,\ i = 1 \\ \bot & \text{otherwise} \end{matrix}\right\} & \text{if } f = var^I \\ (p,i) \mapsto \bot & \text{if } f = var^S, S \neq I \\ i \mapsto \left\{\begin{matrix} (1, C, ()) & \text{if } i = 1 \\ (1, I, ()) & \text{if } i = 2 \\ \bot & \text{otherwise} \end{matrix}\right\} & \text{if } f = cont \\ 3 & \text{if } f = nextchan \\ i \mapsto \left\{\begin{matrix} \mathrm{run} & \text{if } i = -1 \\ \mathrm{fin} & \text{if } i \in \{0,1,2\} \\ \bot & \text{otherwise} \end{matrix}\right\} & \text{if } f = proc \\ 3 & \text{if } f = nextpid \end{cases}$$

Both example formulas are interpreted as 1:

- $\overline{\mathrm{val}}_{g_0}\big([\mathrm{RUN\ GLOBAL}]\ Y_1^I =^I c_{42}\big) \ = \ 1$
  because in g_{11}, which is the only final state, Y_1^I is interpreted as 42.
- $\overline{\mathrm{val}}_{g_0}\big(\langle\langle \mathrm{RUN\ GLOBAL}\rangle\rangle\ \exists x_1^C\ read^I(cont(x_1^C), c_1) =^I c_{42}\big) \ = \ 1$
  because there is an intermediate state and a channel such that the queue indexed by this channel has (non-zero length, sort I and) first element 42, or formally: $\overline{\mathrm{val}}^{\alpha}_{g_8}(cont(x_1^C)) = (1, I, (42))$ with $\alpha(x_1^C) = 2$.

Both formulas are also interpreted as 1 in all other states of the standard structure in which *nextpid* and *nextchan* are not undefined, because the processes only access

- their local variables,
- queues that are constructed during the execution,
- the global variable Y_i^I, which is not read.

Further examples for formulas that hold in the origin, but not in all states are

$$[\mathrm{RUN\ GLOBAL}]\ \forall x_1^C\ \big(\neg cont(x_1^C) =^Q \bot^Q \rightarrow lengthc(x_1^C) =^I c_0\big)$$

which states that all defined queues are empty after termination and

$$[\mathrm{RUN\ GLOBAL}]\ nextpid = c_3.$$

8.3.2 Other Examples

Execution of Two Program in a Row Let $\pi = \pi_0$ be the program of the previous example and let

$$\pi' = \mathrm{RUN\ init2}\ p$$

where $p(I, 1) = Y_1^I\ \boxed{1}$.

And let init2 $\in \mathbf{P}$ be given by

- $NmbVar(\text{init2}, S) = \begin{cases} 1 & \text{if } S = C \\ 0 & \text{otherwise} \end{cases}$
  for $S \in \Sigma_P$
- $Ini(\text{init2})(C, 1) = ()^{2,I}$.
- $NmbPar(\text{init2}, S) = \begin{cases} 1 & \text{if } S = I \\ 0 & \text{otherwise} \end{cases}$
  for $S \in \Sigma_P$
- $Body(\text{init2}) =$
  $X_1^C \mathbin{!} X_1^I - 1$
  $\text{END}(\text{init2}, nextpid)$

Then

$$\overline{\text{val}_O}\big(\langle\langle\pi\rangle\rangle[\pi']\exists x_1^C\ read^I(cont(x_1^C), c_1) =^I c_{40}\big) \;=\; 1$$

holds because for example g_3 is a state in a trace in $\overline{\text{val}_O}(\pi)$ and executing π' from it leads to a single final state g_6' in which the interpretation of the primary non-rigid function symbols is given by

$$\text{val}_{g_6'}(f) = \begin{cases} (p,i) \mapsto \left\{\begin{matrix} 41 & \text{if } p = -1,\ i = 1 \\ \bot & \text{otherwise} \end{matrix}\right\} & \text{if } f = var^I \\ (p,i) \mapsto \left\{\begin{matrix} 1 & \text{if } p = 0,\ i = 1 \\ 2 & \text{if } p = 0,\ i = 2 \\ 0 & \text{if } p = 1,\ i = 1 \\ 1 & \text{if } p = 1,\ i = 2 \\ \bot & \text{otherwise} \end{matrix}\right\} & \text{if } f = var^C \ \boxed{2} \\ (p,i) \mapsto \bot & \text{if } f = var^S, S \notin \{I, C\} \\ i \mapsto \left\{\begin{matrix} (1, C, ()) & \text{if } i = 1 \\ (1, I, ()) & \text{if } i = 2 \\ (2, I, (40)) & \text{if } i = 3 \ \boxed{3} \\ \bot & \text{otherwise} \end{matrix}\right\} & \text{if } f = cont \\ 4 & \text{if } f = nextchan \\ i \mapsto \left\{\begin{matrix} \text{run} & \text{if } i \in \{-1, 0, 1\} \ \boxed{4} \\ \text{fin} & \text{if } i = 2 \ \boxed{5} \\ \bot & \text{otherwise} \end{matrix}\right\} & \text{if } f = proc \\ 3 & \text{if } f = nextpid \end{cases}$$

And $\overline{\text{val}_{g_6'}^{\alpha}}(cont(x_1^C)) = (2, I, (40))$ if $\alpha(x_1^C) = 3$.

$\boxed{1}$ The execution of π' depends on a global program variable.

$\boxed{2}$ The local program variables of init and A are still defined. The local program variables of init2 have been reset by the end command.

$\boxed{3}$ This queue has been created and sent to by init2.

$\boxed{4}$ The states of init and A are still run.

$\boxed{5}$ The state of init2 is fin.

Arbitrary Initial States Let again $\pi = \pi_0$ be the program of the previous example and let

$$g_0 = \big((nextpid, (), \bot)(nextchan, (), \bot)\big) * \text{val}_O$$

be the state in which all primary non-rigid function symbols are undefined. Then *nextpid* and *nextchan* stay undefined throughout the execution of π, all program variables are created for the process ID $\bot$, and all queues are created for the channel $\bot$.

Then $\overline{\mathrm{val}_{g_0}}(\pi)$ contains a single trace of length 5. The last state is g_4'' with

$$\mathrm{val}_{g_4''}(f) = \begin{cases} (p,i) \mapsto \left\{ \begin{matrix} 41 & \text{if } p = \bot,\ i = 1 \\ \bot & \text{otherwise} \end{matrix} \right\} & \text{if } f = var^I \\ (p,i) \mapsto \bot & \text{if } f = var^S, S \neq I \\ i \mapsto \left\{ \begin{matrix} (1, I, ()) & \text{if } i = \bot \\ \bot & \text{otherwise} \end{matrix} \right\} & \text{if } f = cont \\ \bot & \text{if } f = nextchan \\ i \mapsto \left\{ \begin{matrix} \mathrm{run} & \text{if } i = \bot \\ \bot & \text{otherwise} \end{matrix} \right\} & \text{if } f = proc \\ \bot & \text{if } f = nextpid \end{cases}$$

And in this state a timeout occurs because two commands of π_4 (as in the above example) try to receive from an empty queue, and the other command tries to send a channel value to an integer queue. The latter happens because the first queue of the sort channel is overwritten by the second queue of the sort integer because *nextchan* and *incchan*(*nextchan*) have the same semantics if *nextchan* is undefined.

Therefore preconditions should at least require that *nextpid* and *nextchan* are defined in order to avoid unexpected program behavior. This is done in the example derivation of section 10.7.1.

9 Differences to Promela

The programming language that is part of DLTP is closely related to the process meta language Promela. But several details of the Promela specification are different. There are three main reasons for that:

- Advanced (but in most cases easily definable) concepts are omitted.
- Provisions for the model-checker Spin are omitted.
- Weaknesses of Promela are remedied.

In this section the term DLTP refers to the standard structure for the standard vocabulary of DLTP.

9.1 Advanced Concepts

The following advanced concepts of Promela are omitted.

- goto commands: Intuitively, Promela is defined to model automata, which explains the presence of labels and goto commands. DLTP models sequential execution, which explains the usage of the $\mathrm{rem}^{\mathrm{A}}$ and $\mathrm{rem}^{\mathrm{S}}$ mappings that give the remaining program.

  Using goto is convenient, but not elegant and incompatible to sequential execution. In DLTP automata must be modelled by a process that consists of only one do command such that

  - one local integer variable X contains the state of the automaton,
  - the first command of each option is executable if X has a certain value (The binary operator $=$ can be defined by the operations present in DLTP.),
  - anywhere during the execution of an option the next state is stored in X.

- unless commands: unless commands are composed commands that consist of two command sequences[37]. Normally the first sequence is executed, but if the first command of the second sequence becomes executable the execution continues with the second sequence and the first sequence is not executed anymore. unless commands can be used to trap exceptions or (by nesting them) to model a selection where the options have different priorities. Simulating these commands in DLTP is very awkward, but DLTP can be easily extended to include them (see section 13.1.3).

- print and stdin commands: These commands are for user interaction only.

[37]Actually Promela only allows two single commands instead of command sequences, but that is not less general: The single command can be of the type if and contain only one option.

- active declarations: These declarations specify that certain instances of certain processes are present immediately when the execution starts. active declarations can be simulated by instantiating these processes explicitly in the process that declares the global variables.

- Hidden variables and _: Hidden variables and the write-only variable _ are not part of the global system state. Instead of them normal program variables can be used (and ignored when making statements about the state).

- The timeout concept: timeout is a predefined global variable that is set to 1 if no command is executable, through which some commands may become executable. A timeout occurs in Promela only if no command is executable and timeout is already 1. This two-stage timeout cannot be simulated in DLTP[38] where a timeout occurs immediately, but to define an appropriate extension of DLTP is simple.

- Array expressions: Array expressions can be simulated in two ways. Firstly, composed sorts allow array functionality except that the index may not be an expression. Secondly, if the restrictions of variable references (precisely: of the second argument of var^S for $S \in \Sigma_P$) in process type bodies (Definition 11) were eased, the local variables of a fixed sort would become one big array; however, this would make update substitutions inefficient[39] (see section 13.4).

- The data type mtype and user defined data types: These can easily be simulated by the data type I.

- The data types bit, bool, byte, short and int, type casts and overflows: The data types are all contained in the single data type I of DLTP. If overflow effects are necessary they can be simulated. It is also possible to use a more complex vocabulary to have all the Promela data types and type casts (see section 13.1.1).

- Logical and comparison operators: All operators of Promela can be defined by the arithmetical operators of DLTP (In particular comparisons are integer expressions that return 1 or 0.). This is not obvious, but it follows from the semantics of division: If $sgn(t)$ abbreviates

  $$\left[t =^I ini^I \;?\; 0 \;:\; [t-1 =^I ini^I \;?\; 1 \;:\; (t+1)/t]\right]$$

  for $t \in Terms(I)$, then $\overline{\mathrm{val}}^{\alpha}_{g}(sgn(t)) = \begin{cases} 1 & \text{if } \overline{\mathrm{val}}^{\alpha}_{g}(t) > 0 \\ 0 & \text{otherwise} \end{cases}$

  Then all operators of Promela can be introduced as abbreviations of DLTP terms, for example $t > t'$ as an abbreviation of $sgn(t-t')$ for $t, t' \in Terms(I)$.

- The predefined constants TRUE, FALSE and SKIP of Promela can be replaced with c_0 and c_1 in DLTP.

Of these omitted concepts only unless and timeout cannot be intuitively simulated in DLTP and require extensions of the definition.

9.2 Model-Checking

The following concepts are included in Promela because the model-checker and simulator Spin uses them to allow or simplify the verification. They are superfluous if a logical approach is taken and are therefore omitted.

- The never claim: The never process is a special process that is scheduled before every execution of a command of another process. It is considered a violation of a temporal statement on the system behavior if the never process terminates. Spin contains an algorithm that generates the body of the never process from a negated LTL formula.

- The following special expressions can be used in the body of the never process to refer to the system state: np_ (true if no process is in a local state with a label that starts with "progress"), enabled (true if the next command of a certain process is executable), pc_value (returns the program counter of a process), @ (allows to check whether a certain process is in a certain local state), _last (gives the process ID of the last process from which a command was executed) and process identification numbers as return values of run expressions (needed as an argument of enabled, pc_value and @).

[38] At least it cannot be intuitively simulated. Of course DLTP is Turing complete.
[39] If the restriction were lifted the results on the calculus **C** would still hold.

- As a side effect of the definition of rem^A the process ID of a process can be accessed by the process itself, using the function symbol *nextpid*. During the execution this function symbol is replaced with other terms to ensure that it is always interpreted as the process ID. This corresponds to the local variable _pid of Promela.
- Labels that start with "accept" can be used inside the never claim to specify that the labelled command may only be executed finitely often.
- trace and notrace declarations: Another way to specify correctness claims besides the never process.
- assert commands: assert commands are always executable and have no effect, but if the expression specified in an assert command is 0 this is reported.
- d_step commands: d_step can be used instead of ATOMIC if the atomic sequence is deterministic. This improves the efficiency of the model-checking.
- Exclusive channel access declarations: xr and xs can be used to declare that one process has exclusive read or write access to a channel. This is used to improve the efficiency of the model-checking.
- Channel status polls: For efficiency the tests whether a channel is full or empty may not be negated in Promela. Instead the functions full, empty, nfull and nempty are provided. For DLTP *lengthc* is sufficient.
- provided constraints: A provided clause can be used as part of a process declaration. It is used to restrict the executability of any command of the process. This is usually only used in simulations.
- priority declarations of processes: These are provided for simulations. Processes with higher priorities are more likely to be scheduled.

9.3 Weaknesses

9.3.1 Processes

In Promela the number of processes that are active at the same time must be a priori finitely bounded. This constraint can be lifted in DLTP, and this is the most important advantage the logical approach has over the model-checking approach. Therefore in DLTP run commands are always executable.

9.3.2 Domain of Computation

In DLTP the full set $\mathbb{Z}$ is used whereas Promela is restricted to a finite subset of the integers. This is a normal advantage that the logical approach has over the model-checking approach.

It should be noted that the use of all natural numbers is essential for the completeness proof since only thus Gödelization is possible.

9.3.3 Channels and Queues

Motivation The specification of the Promela semantics is confusing because several constraints on data types in channel operations are not clear and are violated in the Spin interpreter anyway.

In Promela channel variables are implicitly pointers to a non-specified data type that holds the content of the channel. In order to obtain a clear specification of the semantics of channels queues are introduced.

It would also be possible to combine channels and queues into one data type: Then the channel is a queue as defined here with an ID. But this has two disadvantages: The whole queue must be allowed as an element of another queue[40]; and the formation of queue terms outside programs is more complicated.

[40] In Spin ([2]) and [7] only the ID is an element of the queue, but that is not formally justified. In [7] it is called an "ugly hack" (p. 8).

Channels Only channel terms occur in programs. They are interpreted as pointers to queues and are dereferenced by the function symbol *cont*. Channels can be passed between processes as parameters or via channels. Direct assignments to channel variables are possible in DLTP, but the only program terms of the sort C are other channel variables.

In Promela direct assignments to channels are not allowed, but indirect assignments by receiving a channel from another channel can be used both in Promela and in DLTP.

Therefore in Promela and in DLTP a process can access a queue only in certain cases: If it created the queue itself, if it is passed a channel pointing to this queue as a parameter, or if it can access a different queue that holds a channel pointing to the queue and so on.

In Spin ([2]) even direct assignments to channels are possible in the following form: $Chan\ C; C = n + 0;$ where n is any integer expression. This would be a syntax error in Promela and DLTP. Therefore in Spin ([2]) every process can access every channel.[41]

Queues Queues store the elements that are passed to a channel. A queue is a triple of capacity, sort, and a sequence over the universe of that sort with a length that is at most the capacity. Queues of capacity 0 are accessed synchronously and never contain an element.

Programs modify queues implicitly by channel operations: Initialization of a channel variable constructs a new queue, and send and receive commands access the queue that is pointed to by the channel variable. If a channel variable is undefined or points to the undefined queue commands that send to or receive from this channel are not executable. There is no garbage collection. A queue is not destructed if no program variables and no queues hold a reference to that queue anymore.

Outside programs, terms of the sort Q can be built up using the function symbols $()^{m,S}$, $insert^S$ and $remove$ for $m \in \mathbb{N}$ and $S \in \Sigma_P$.

Type Correctness It is not possible to decide syntactically whether the sort of a $\underline{\text{send}}$ or $\underline{\text{receive}}$ command matches the sort of the run-time dependent queue the channel points to. In Spin ([2]) this problem is ignored: Internally all data types in a queue are integers and they are cast to the needed data type when a $\underline{\text{receive}}$ command is executed. In particular it is possible to pass an integer expression to a channel and read it onto a channel variable.

This effect is most probably not wanted by the designers of both Promela and Spin. However, it must be noted that this is not a mere bug in the Spin ([2]) implementation, but rather a fundamental design flaw of Spin.

In Promela, to ensure type correctness, it is syntactically forbidden to use a $\underline{\text{send}}$ or $\underline{\text{receive}}$ command with a sort that does not match the sort of the queue pointed to by the channel variable, although this can only be decided at run-time. In DLTP this is formally clarified by defining that such a command is allowed, but not executable.

9.3.4 Traces

In certain situations Promela does not specify precisely which states are part of a trace.

In DLTP this impreciseness is resolved in the following way:

- The initial state, in which no command has been executed yet, is part of the trace.
- When a process is instantiated, its local variables (which are the global variables if the value of *nextpid* is -1) are initialized in the effect of the $\underline{\text{run}}$ command. That means that there is one state in which the process is not running yet, and in the next state it is running and all its variables are initialized.
- When the last command of a process has been executed and its effect applied, an additional state is added to the trace, in which the local variables of the terminated process are reset and the process state is set to fin, but nothing else is changed. This state is the effect of a command of the type $\underline{\text{end}}$.

In Spin these impreciseness may be resolved differently which would lead to a different trace definition. Therefore some formulas of DLTP may be interpreted differently from the interpretation of the $\underline{\text{never}}$ claim by Spin.

[41] Forbidding these direct assignments in Spin does not alleviate the problem because Spin cannot prevent type casts from integer to channel during indirect assignments.

It would be possible to derive the specification of the trace definition by reverse-engineering from the Spin implementation. This is not done in this work[42] because the Spin implementation is no formal specification and subject to change anyway.

9.3.5 Incomplete Specification

Division Promela does not specify the result of a division operation if the precise result is a fraction. In DLTP the result is rounded to zero because that is what Spin ([2]) does.

Division by zero returns the result $\perp$ in DLTP. It would also be possible to make commands that try to divide by zero not executable. In Spin ([2]) the problem is ignored and it even crashes in such a situation.

Comparison of Channels Promela does not specify the result of a comparison between channels. The definition used in DLTP is chosen for compatibility with Spin ([2]).

9.3.6 Pattern Matching

The pattern matching of Promela originally allowed only constant expressions as patterns. In later versions the content of variables could be used as a pattern by using the keyword EVAL.

In DLTP any expression can be used as a pattern. These expressions are evaluated during the executability check of the synchronous transfer.

9.4 Other Differences

Process Declarations and Program Counters Promela programs essentially consist of the process type declarations and some information which processes are initially instantiated when the program is executed. During the execution the active process instances and their program counters are part of the global system state. This is not possible in a dynamic logic: The active processes along with their code would have to be stored as the values of non-rigid function symbols.

In DLTP a program consists of the remaining code of the active processes. The remaining-program mappings and the EC and BT labels are used instead of program counters and execution control information. For convenience the process type declarations are not part of the program, but are given in some context.[43]

It is possible to give DLTP programs that contain processes with code that does not satisfy the restrictions of process type bodies in Definition 11: Any program variable, as for example in the command

$$var^C(c_1, c_1) := var^C\big(var^I(c_2, c_1), c_1\big),$$

can occur in these processes. This may sometimes be desirable, but in most cases it should be avoided, e. g. by sensible program execution (see the remark in section 8.2).

Program Variables In Promela variable declarations can occur anywhere in the body of a process type, but are executed instantly upon instantiation. In DLTP these declarations are separated from the body and captured by the mappings $NmbVar$ and Ini. This is a purely syntactical difference.

In Promela global variables are declared independently of process types. It would be possible to do this in DLTP, too, but it is more elegant to use a special process (that is not terminated by an end command) and to define its local variables to be the global variables; this is the process with ID -1. It is convenient to use the body of this process to instantiate the first real process, e. g. the init process.

The names of program variables are completely different in DLTP. Also global and local variables are syntactically distinguished in DLTP. The translation is simple.

[42]The first two open questions are resolved in Spin ([2]) as in DLTP.

[43]Formally, this context is fixed once and for all in Definition 11; but in practice the needed process types are represented by symbols, the meaning of which is declared explicitly as in the example in section 4.4.

Composed sorts The handling of composed sorts is completely different, but equally expressive. In Promela the components of the sort are read and written individually, and the operator . is used to separate the components of a variable reference. In DLTP the function symbols $elem^S$ and $tuple^S$ are used and a variable of a composed sort is always read and written as a whole.

For example if the composed sort S is given by

$$\text{typedef } S \; \{\text{int } C1; \; \text{int } C2\}$$

(which corresponds to $S = I \times I$ in DLTP), the assignment $X.C1 := t$ of Promela has to be written as $X := tuple^S(t, elem^S_2(X))$ in DLTP.

This entails the following small advantages of DLTP over Promela:

- All composed sorts are allowed as parameters of processes.
- All composed sorts are allowed as commands of the type expression.
- All composed sorts can be initialized.

However, unlike Promela DLTP cannot distinguish between the composed sorts

$$\{S \; C1; \; \text{int } C2\} \text{ and } \{\text{int } C1; \; S \; C2\}$$

where S is as above (Spin does not distinguish between these sorts either.). But this distinction is purely syntactical anyway because both the componentwise composition of equality (needed for pattern matching) and the lexicographic composition of orderings (needed for sorted sending) are associative.

Channel Sorts In Promela a channel does not have a single sort, but a list of sorts, and the corresponding tuples of objects can be stored in the channel. This is superfluous in DLTP since all composed sorts exist in DLTP without being declared explicitly. For example instead of a Promela channel that holds messages of the form $\{S, \text{int}\}$ (where S is as above) a DLTP channel that points to a queue of the sort $I \times I \times I$ can be used.

Processes There are several minor differences in the treatment of processes:

- In DLTP the instantiation of a new process is a special type of command, not an expression as in Promela. And the process ID is not returned to the instantiating process.
- The termination or deconstruction of a process is done explicitly by an end command in DLTP that is executed immediately after the last command of the process. In Promela processes terminate implicitly when all their child processes have terminated.
- There is a confusing remark in the Promela specification[44] that states that new processes have local copies of the variables of the instantiating process. This is probably an error, since neither the Promela syntax nor Spin allow references to these variables. This is therefore not part of DLTP.
- In Promela processes may only terminate if their child processes have terminated. This constraint is lifted in DLTP, since it has no relevant effect (other than a slightly different trace definition because in Promela programs local variables of terminated processes disappear from the global system state at a later time than in DLTP) if the number of processes is unbounded.
- The process ID of a terminated process is not reused in DLTP.

Undefined Values DLTP specifies how undefined values are handled in the computation. In Promela undefined values cannot occur, because the problem of division by zero is ignored and because every used program variable must be declared explicitly (see the remark in section 8.2).

Channel Polls The syntax of the poll commands that check the executability of a receive command is different, but equivalent in DLTP and Promela.

[44]This remark is in [1], in the first note in the section Declarators - Datatypes.

Else Options In Promela there may only be one else options in any process state. That means that for $c \in \underline{\text{if}} \cup \underline{\text{do}}$

$$optelse\_total(c) = \sum_{\substack{i \in [1;opt(c)],\\ c[i]_1 \in \underline{\text{if}} \cup \underline{\text{do}}}} optelse\_total(c[i]_1) + optelse(c)$$

must be at most 1. This restriction is lifted in DLTP.

Break commands In Promela a break command may not occur outside a do command. This restriction is lifted in DLTP: Such a break command terminates the process.

10 A Sequent Calculus for the Standard Structure

In this section at first the basic definitions of sequent calculi are given. In the middle part of the section the rules of a specific sequent calculus are listed, grouped into several categories. Then some example derivations are shown. The global soundness and completeness of this calculus is proven in later sections.

10.1 Preliminary Definitions

10.1.1 Sequents

Syntax

Definition 43. *A* sequent *is pair of two finite sets of formulas. The sequent* (Γ, Δ) *is written* $\Gamma \vdash \Delta$. *In this notation* Γ *or* Δ *is omitted if it is empty. And the set* $\Gamma_1 \cup \ldots \Gamma_m \cup \{F_1, \ldots, F_n\} \subseteq Form$ *is written* $\Gamma_1, \ldots \Gamma_m, F_1, \ldots, F_n$ *where the order is arbitrary.*

Semantics An interpretation function val_g can be extended to sequents relative to an assignment α. This extension is also denoted by $\overline{\text{val}_g^\alpha}$.

Definition 44. *For the interpretation function* val_g *and the assignment* α *the* semantics of a sequent $S = F_1, \ldots, F_m \vdash G_1, \ldots, G_n$ *with formulas* $F_1, \ldots, F_m, G_1, \ldots, G_n$ *is defined by*

$$\overline{\text{val}_g^\alpha}(S) = \overline{\text{val}_g^\alpha}\big((F_1 \wedge \ldots \wedge F_m) \rightarrow (G_1 \vee \ldots \vee G_n)\big).$$

Instead of the dependence on the assignment the following definition is often used:

$$\overline{\text{val}_g}(S) = \begin{cases} 1 & \text{if } \overline{\text{val}_g^\alpha}(S) = 1 \text{ for all assignments } \alpha \\ 0 & \text{otherwise} \end{cases}$$

Elementary Properties For all formulas F, all interpretation functions val_g and all assignments α,

$$\overline{\text{val}_g^\alpha}(F) = \overline{\text{val}_g^\alpha}(\ \vdash F) = \neg\overline{\text{val}_g^\alpha}(F \vdash\ ).$$

For all sequents $S = F_1, \ldots, F_m \vdash G_1, \ldots, G_n$ with formulas $F_1, \ldots, F_m, G_1, \ldots, G_n$, all interpretation functions val_g and all assignments α:

$$\overline{\text{val}_g^\alpha}(S) = \sup\left(\left\{\neg\overline{\text{val}_g^\alpha}(F_i) | i \in [1;m]\right\} \cup \left\{\overline{\text{val}_g^\alpha}(G_i) | i \in [1;n]\right\}\right)$$

10.1.2 Rules

Definition 45. *A* rule *is a pair of a finite set of sequents, called the premises, and a sequent, called the conclusion. The rule* $(\{P_1, \ldots, P_n\}, C)$ *is written*[45]

$$\frac{P_1 \ + \ \ldots \ + \ P_n}{C}$$

In the case $n = 0$ *the rule is called an* axiom.

A calculus *is a set of rules.*

Unless mentioned otherwise the context of a rule is omitted. That means that for $A_1, \ldots, A_n, S_1, \ldots, S_n \subseteq \mathit{Form}$ the rule

$$\big(\{(A_1 \cup \Gamma, S_1 \cup \Delta), \ldots, (A_n \cup \Gamma, S_n \cup \Delta)\}, (A \cup \Gamma, S \cup \Delta)\big) \text{ for finite sets } \Gamma, \Delta \text{ of formulas}$$

is abbreviated by

$$\big(\{(A_1, S_1), \ldots, (A_n, S_n)\}, (A, S)\big).$$

Definition 46. **C** *is a distinguished calculus. It consists of the rules* (R 1) *to* (R 151) *which are given in this section.*

10.1.3 Proofs

Definition 47. *For a sequent S, a set of sequents* **S** *and a calculus C, S is* derived *from or* reduced *to* **S** *in C, if there is a finite tree over sequents with root S and the property that every node N (except for possibly leafs that are elements of* **S***) is the conclusion of a rule of C the premises of which are precisely the children of N.*

If $\mathbf{S} = \emptyset$*, the tree is called a* proof *of S in C.*

10.1.4 Soundness and Completeness

Definition 48. *Let K be a structure with set of states G, let G' be a subset of G and let C be a calculus.*

- *A rule $(\{P_1, \ldots, P_n\}, Q)$ is* (strongly) sound *for K and G' if $\overline{\mathrm{val}}_g(P_i) = 1$ for all $g \in G'$ and all $i \in [1; n]$ implies (is equivalent to) $\overline{\mathrm{val}}_g(Q) = 1$ for all $g \in G'$.*
- *C is (strongly) sound for K and G' if all its rules are (strongly) sound for K.*
- *In the case $G' = G$ the (strong) soundness for K and G' is called global (strong) soundness for K. And in the case $G' = \{g\}$ the (strong) soundness for K and G' is called local (strong) soundness for K and g.*
- *C is* complete *for the structure K, for G' and for the set of sequents* **S***, if for every sequent $S \in \mathbf{S}$ such that $\overline{\mathrm{val}}_g(S) = 1$ for all $g \in G'$ there is a proof of S in C.*

  In the case where **S** *consists of all sequents, the reference to* **S** *is omitted.*

Remark These definitions of soundness and completeness are more general than the usual definitions: They are relative to a set of states $G' \subseteq G$. The usual definitions arise in the special case where $G' = G$.

10.2 Structural Rules and First-Order Predicate Logic

For all formulas F:

Self-proof

$$\frac{}{F \vdash F} \tag{R 1}$$

Case distinction or cut

$$\frac{F \vdash \quad + \quad \vdash F}{\vdash} \tag{R 2}$$

Weakening

$$\frac{\vdash}{\vdash F} \tag{R 3}$$

This rule is not needed for any completeness result.

[45]It is not common to use a special symbol to separate the premises of a rule. + is used here because it enhances readability significantly compared to the usual whitespace character.

Propositional Logic For all formulas F and G:

$$\frac{}{\vdash \ true} \tag{R 4}$$

$$\frac{\vdash}{true \ \vdash} \tag{R 5}$$

$$\frac{\vdash}{\vdash \ false} \tag{R 6}$$

$$\frac{}{false \ \vdash} \tag{R 7}$$

$$\frac{F \ \vdash}{\vdash \ \neg F} \tag{R 8}$$

$$\frac{\vdash \ F}{\neg F \ \vdash} \tag{R 9}$$

$$\frac{\vdash \ F \ + \ \vdash \ G}{\vdash \ F \wedge G} \tag{R 10}$$

$$\frac{F \ , \ G \ \vdash}{F \wedge G \ \vdash} \tag{R 11}$$

$$\frac{\vdash \ F \ , \ G}{\vdash \ F \vee G} \tag{R 12}$$

$$\frac{F \ \vdash \ \ + \ G \ \vdash}{F \vee G \ \vdash} \tag{R 13}$$

Quantifiers For all formulas F:

For $x \in LV(S)$ and

- $S \in \Sigma_E \cup \Sigma_L$, $\Gamma, \Delta \subseteq Form$ and if $X \in LV(S)$ is a logical variable that does not occur free in any formula in $\Gamma \cup \Delta$ or bound in F (These rules are stated with context.):

$$\frac{\Gamma \ \vdash \ \{X/x\}F, \Delta}{\Gamma \ \vdash \ \forall x \ F, \Delta} \tag{R 14}$$

$$\frac{\Gamma, \{X/x\}F \ \vdash \ \Delta}{\Gamma, \exists x \ F \ \vdash \ \Delta} \tag{R 15}$$

- $S \in \Sigma_P \setminus \Sigma_E$ and if $X_i \in LV(S_i)$ does not occur in F for $i \in [1; n]$:

$$\frac{\vdash \ \forall X_1 \ldots \forall X_n \ \{tuple^S(X_1, \ldots, X_n)/x\}F}{\vdash \ \forall x \ F} \tag{R 16}$$

$$\frac{\exists X_1 \ldots \exists X_n \ \{tuple^S(X_1, \ldots, X_n)/x\}F \ \vdash}{\exists x \ F \ \vdash} \tag{R 17}$$

For $x \in LV(S)$, $S \in \Sigma$ and a term $t \in Terms(S)$:

$$\frac{\forall x \ F, \{t/x\}F \ \vdash}{\forall x \ F \ \vdash} \tag{R 18}$$

$$\frac{\vdash \ \exists x \ F, \{t/x\}F}{\vdash \ \exists x \ F} \tag{R 19}$$

Renaming of Bound Variables For $S \in \Sigma$, a formula F that contains a bound logical variable $x \in LV(S)$ in a subformula $Qx\ G$ of F for $Q \in \{\forall, \exists\}$ and a formula G, and that does not contain the logical variable $y \in LV(S)$:[46]

$$\frac{\vdash\ F'}{\vdash\ F} \qquad \text{(R 20)}$$

$$\frac{F'\ \vdash}{F\ \vdash}$$

where F' denotes F with some occurrence of the word $Qx\ G$ replaced with $Qy\ \{y/x\}G$.

This rule is only needed to allow to apply the rules (R 53) and (R 54).

Application of Equality If

- U is a sequence of updates,
- F is a formula,
- $S \in \Sigma$ and $t, t' \in Terms(S)$
- F' arises from F by replacing some occurrences of t in F with t'
- either F is first-order, or $F = u\ G$ for an update u and a formula G and all replaced occurrences of t are in u

then:[47]

$$\frac{U\ t = t'\ \vdash\ U\ F'}{U\ t = t'\ \vdash\ U\ F} \qquad \text{(R 21)}$$

$$\frac{U\ t = t'\ ,\ U\ F'\ \vdash}{U\ t = t'\ ,\ U\ F\ \vdash}$$

Conditional terms For a sequence of updates U and a term $T = [C?t : t'] \in Terms(S)$:

$$\frac{U\ T =^S t\ ,\ U\ C\ \vdash}{U\ C\ \vdash} \qquad \text{(R 22)}$$

$$\frac{U\ T =^S t'\ \vdash\ U\ C}{\vdash\ U\ C} \qquad \text{(R 23)}$$

U could be left out from the rules, but this stronger version of the rules is more efficient.

10.3 Elimination of Programs

The rules for the elimination of programs are the most characteristic rules of the given calculus. They are also the most complex rules. After some preliminary definitions they are given in several groups.

Principal Idea The idea of the rules is as follows:

- The rules can be applied if a proof goal in the succedent has the form $U\ M(\pi)$. This situation is reached by using the rules for first-order logic, updates and duality of modal operators. Then π is executed symbolically and further updates are added to U to the right-hand side.

[46] For simplicity the two rules are referenced with the same number.
[47] For simplicity the two rules are referenced with the same number.

- During the scheduling all possibilities from which process the next elementary command or commands can be chosen are considered and encoded by the tags. For each possibility the proof branches (if $M \in \mathit{Oper}^U$) or offers alternative proof goals (if $M \in \mathit{Oper}^E$). This phase corresponds to the definition of $\overline{\mathrm{val}_g}(\pi)$ relative to the sets $\overline{\mathrm{val}_g}(i:\ \pi)$ and $\overline{\mathrm{val}_g}(i,j:\ \pi)$.
- During the unwinding the composed commands at the beginning of those (one or two) processes indexed by the tags are unwound so that the processes start with elementary commands. Unwinding selections leads to further branches or alternatives in the proof tree. This phase corresponds to the computation of $\mathrm{Unwind}^{\mathrm{A}}$ and $\mathrm{Unwind}^{\mathrm{S}}$.
- During the symbolic execution the executability of the elementary commands at the beginning of those processes indexed by the tag is checked. If the commands are not executable the proof goal is closed (if $M \in \mathit{Oper}^U$) or simplified (if $M \in \mathit{Oper}^E$). This corresponds to the executability check in the definition of $\mathrm{Unwind}^{\mathrm{A}}$ and $\mathrm{Unwind}^{\mathrm{S}}$. If the commands are executable their effect is applied and the program is replaced with the remaining program. This corresponds to the definition of $\overline{\mathrm{val}_g}(i:\ \pi)$ and $\overline{\mathrm{val}_g}(i,j:\ \pi)$.
- The symbolic execution rules remove the tags and the scheduling takes place again.
- If the program in the modal operator is empty the modal operator is removed.
- If the program does not terminate the modal operator can only be eliminated by the Gödelization rules (or by other rules that eliminate the whole formula).
- If all modal operators have been removed the updates are applied to eliminate the non-rigid function symbols.

Statement of the Applicability Conditions All rules can only be applied under certain conditions that can be syntactically checked. It is sometimes necessary to introduce new variables as abbreviations for the objects postulated by these conditions. These new variables can be subject to further applicability conditions.

To enhance readability the following convention is followed: The introduction of independent variables and the applicability conditions are stated using the symbols $=$ and $\in$. The dependent variables are introduced in equalities that use the sign $:=$.

The following variables are introduced here and apply to all rules in this section:

- U is a finite sequence of updates.
- $M \in \mathit{Oper}$ is a modal operator.
- $\pi \in \mathit{Prog} \setminus \{\epsilon\}$ is a non-empty program.
- $n := \mathrm{Length}(\pi)$
- $F \in \mathit{Form}$ is a formula.

10.3.1 Preliminary Definitions

Asynchronous Executability

Definition 49. *Exec^A assigns a formula with every command:*

$$\mathit{Exec}^A(c) = \begin{cases} \neg t =^S \mathit{ini}^S & \text{if } c \in \underline{\text{expression}},\ c = t,\ t \in \mathit{Terms}(S) \\ \mathit{lengthc}(C) + c_1 \leq^I \mathit{cap}(\mathit{cont}(C)) & \\ \qquad \wedge \mathit{sor}^S(\mathit{cont}(C)) & \text{if } c \in \underline{\text{send}},\ C = \mathit{Chan}(c), S = \mathit{Sor}(c) \\ \mathit{pollfirst}^{\sigma,S}(\mathit{Chan}(c), a_{i_1}, \ldots, a_{i_r}) =^I c_1 & \text{if } c \in \underline{\text{receive}},\ \mathit{Type}(c) \in \{?, ?<\}, \\ & \qquad \mathit{Pos}(c) = \sigma = \{i_1, \ldots, i_r\}, a = \mathit{Arg}(c) \\ \mathit{pollany}^{\sigma,S}(\mathit{Chan}(c), a_{i_1}, \ldots, a_{i_r}) =^I c_1 & \text{if } c \in \underline{\text{receive}},\ \mathit{Type}(c) \in \{??, ??<\}, \\ & \qquad \mathit{Pos}(c) = \sigma = \{i_1, \ldots, i_r\}, a = \mathit{Arg}(c) \\ \bigvee_{i=1}^{\mathit{opt}(c)} \mathit{Exec}^A(c[i]_1) & \text{if } c \in \underline{\text{if}} \cup \underline{\text{do}},\ \mathit{optelse}(c) = 0 \\ \mathit{Exec}^A(c[\mathit{body}]_1) & \text{if } c \in \underline{\text{atomic}} \\ \mathit{true} & \text{otherwise} \end{cases}$$

The intuitive meaning of $Exec^A(c)$ is that the command c is asynchronously executable. This is formalized in Lemma 5.

Synchronous Executability

Definition 50. *$Exec^S$ assigns a formula with two commands c and d:*[48]

$$Exec^S(c,d) = \begin{cases} E(c,d) & \text{if } c \in \underline{\text{send}},\ d \in \underline{\text{receive}} \\ \bigvee_{i=1}^{opt(c)} Exec^S(c[i]_1, d) & \text{if } c \in \underline{\text{if}} \cup \underline{\text{do}} \\ \bigvee_{i=1}^{opt(d)} Exec^S(c, d[i]_1) & \text{if } d \in \underline{\text{if}} \cup \underline{\text{do}} \\ Exec^S(c[body]_1, d) & \text{if } c \in \underline{\text{atomic}} \\ Exec^S(c, d[body]_1) & \text{if } d \in \underline{\text{atomic}} \\ false & \text{otherwise} \end{cases}$$

where

$$E(c,d) = \ C =^C C' \ \wedge \ cap(cont(C)) =^I c_0 \ \wedge \ sor^S(cont(C))$$
$$\wedge\, recpos^{\sigma,S'}\big(insert^S\big(()^{1,S}, Arg(c), c_1\big), a_{i_1}, \ldots, a_{i_r}\big) =^I c_1$$

with $C = Chan(c)$, $C' = Chan(d)$, $S = Sor(c)$, $S' = Sor(d)$, $a = Arg(d)$ and $\sigma = \{i_1, \ldots, i_r\}$.

The intuitive meaning of $Exec^S(c,d)$ is that the commands c and d are synchronously executable. This is formalized in Lemma 6.

Effect of Asynchronous Execution

Definition 51. *Eff^A assigns a sequence of updates with every elementary command c. To improve readability the definition is split into several cases:*

- *$c \in \underline{\text{assignment}}$, $c = var^S(t_1, t_2) := t$:*

  $$Eff^A(c) = (var^S, (t_1, t_2), t)$$

- *$c \in \underline{\text{run}}$: Let*
  - $\text{PT} = Proc(c)$ *and* $p = Par(c)$,
  - $MIni(\text{PT})$, n, (C_i, Q_i) *for* $i \in [1; n]$ *and* C_{n+1} *be computed from* PT *by the initialization algorithm (page 42),*
  - $V_S = NmbVar(\text{PT}, S)$ *and* $P_S = NmbPar(\text{PT}, S)$.

  Then $Eff^A(c)$ consists of the following updates[49]*:*
  - $(proc, nextpid, run)$ $\boxed{1}$
  - $\big(var^S, (nextpid, c_i), MIni(PT)(S, i)\big)$ *for* $S \in \Sigma_P$, $i \in [1; V_S]$ $\boxed{2}$
  - $(cont, C_i, Q_i)$ *for* $i \in [1; n]$ $\boxed{3}$
  - $(nextchan, (), C_{n+1})$ $\boxed{4}$
  - $\big(var^S, (nextpid, c_i), p(S, i - V_S)\big)$ *for* $S \in \Sigma_P$, $i \in [V_S + 1; V_S + P_S]$ $\boxed{5}$
  - $(nextpid, (), nextpid + c_1)$ $\boxed{6}$

- *$c \in \underline{\text{send}}$, $C = Chan(c)$:*

  $$Eff^A(c) = (cont, C, insert(cont(C), Arg(c), p))$$

  $$\textit{where } p = \begin{cases} length(cont(Chan(c))) + c_1 & \text{if } Type(c) =! \\ sendpos^S(cont(C), Arg(c)) & \text{if } Type(c) =!! \end{cases} \boxed{7}$$

[48] To make $Exec^S(c,d)$ well-defined, if two options of the case distinction apply, the first one is chosen.

- *$c \in \underline{\text{receive}}$, $Type(c) \in \{?<, ??<\}$, $C = Chan(c)$, $\sigma = Pos(c) = \{i_1, \ldots, i_r\}$, $S = Sor(c)$ and $a = Arg(c)$:*

  $$Eff^A(c) = V$$

  where

  $$p = recpos^{\sigma,S}(cont(C), a_{i_1}, \ldots, a_{i_r})\ \boxed{8}$$

  and V consists of the following updates[49]*:*

  $$\left(var^{S_i}, (s_i, t_i), elem^S_i\left(read^S(cont(C), p)\right)\right)\ \boxed{9}$$

  with $Var(c)_i = var^{S_i}(s_i, t_i)$ for $i \in [1; \mathrm{Length}(S)] \setminus \sigma$.

- *$c \in \underline{\text{receive}}$, $Type(c) \in \{?, ??\}$, $C = Chan(c)$, $S = Sor(c)$:*

  $$Eff^A(c) = V \cdot \left(cont, C, remove(cont(C), p)\right)$$

  where V and p are as above

- *$c \in \underline{\text{end}}$, $c = \mathrm{END}(\mathrm{PT}, t)$: $Eff^A(c)$ consists of the following updates*[49]*:*
  - $(proc, t, fin)\ \boxed{10}$
  - $(var^S, (t, c_i), \bot^S)\ \boxed{11}$ *for $S \in \Sigma_P$ and $i \in [1; NmbVar(\mathrm{PT}, S) + NmbPar(\mathrm{PT}, S)]$.*

- *In all other cases: $Eff^A(c) = ()$*

Effect of Synchronous Execution

Definition 52. *Eff^S assigns a sequence of updates with every pair of elementary commands:*

- *If $c \in \underline{\text{send}}$, $d \in \underline{\text{receive}}$, $\overbrace{Sor(c) = Sor(d)}^{\boxed{12}}$: $Eff^S(c, d)$ consists of the following updates*[49]*:*

  $$(var^{S_i}, (s_i, t_i), Arg(c)_i) \text{ with } Var(d)_i = var^{S_i}(s_i, t_i) \text{ for } i \in [1; n] \setminus Pos(d)\ \boxed{13}$$

- *In all other cases: $Eff^S(c, d) = ()$*

The intuitive meaning of $Eff^A(c)$ and $Eff^S(c, d)$ is that they syntactically give the changes the asynchronous execution of c or the synchronous execution of c and d makes to the state. This is formalized in Lemma 7.

$\boxed{1}$ The process state is changed.

$\boxed{2}$ The local variables that are not parameters are initialized.

$\boxed{3}$ The new queues are constructed.

$\boxed{4}$ The next free channel is updated.

$\boxed{5}$ The parameters are initialized.

$\boxed{6}$ The next free process ID is incremented.

$\boxed{7}$ p gives the position of insertion: It is the end of the queue for normal sending and determined by $sendpos^S$ for sorted sending.

$\boxed{8}$ p gives the position of the element that is retrieved from the queue: It is the first matching element in the queue. This value is 1 for an executable normal <u>receive</u> command.

$\boxed{9}$ V gives the updates that assign the retrieved values to the variables.

$\boxed{10}$ The process state is changed.

$\boxed{11}$ The program variables of the process are reset.

$\boxed{12}$ If this condition were dropped some not executable synchronous transfers would have a syntactically incorrect effect.

$\boxed{13}$ The sent values are assigned to the variables.

Although the effect formulas $Eff^A(c)$ and $Eff^S(c, d)$ are irrelevant if the commands are not executable, they are defined and consist of syntactically correct updates for all possible arguments.

[49] The order is almost arbitrary. To make $Eff^A(c)$ well-defined the order of the listing and ascending order of i are chosen.

10.3.2 Duality

$$\frac{\vdash\ U\ M'(\pi)\neg F}{U\ M(\pi)F\ \vdash} \tag{R 24}$$

10.3.3 Scheduling

Termination of Programs

$$\frac{\vdash\ U\ F}{\vdash\ U\ M(\epsilon)F} \tag{R 25}$$

Termination of Processes For

- $i \in [1; n]$
- $\mathrm{First}(\pi_i) \in \underline{\mathrm{end}}$
- $\mathrm{First}(\pi_j) \notin \underline{\mathrm{end}}$ for $j \in [1; i-1]$

$$\frac{\vdash\ \overbrace{U\ M(i:\ \pi)}^{\boxed{1}}\,F}{\vdash\ U\ M(\pi)F} \tag{R 26}$$

No Exclusive Control and Timeouts For

- $\mathrm{First}(\pi_i) \notin \underline{\mathrm{end}}$ for $i \in [1; n]$
- $\mathrm{Label}(\mathrm{First}(\pi_i)) \neq \mathrm{EC}$ for $i \in [1; n]$
- $\boxed{2}\ Ex = \bigvee_{i=1}^{n} Exec^{A}\big(\mathrm{First}(\pi_i)\big) \vee \bigvee_{\substack{i,j\in[1;n]\\ i\neq j}} Exec^{S}\big(\mathrm{First}(\pi_i), \mathrm{First}(\pi_j)\big)$
- $\boxed{3}\ T := \begin{cases} true & \text{if } M = []\ \boxed{4} \\ false & \text{if } M = \langle\rangle\ \boxed{5} \\ U\ F & \text{otherwise}\ \boxed{6} \end{cases}$

If $M \in Oper^{U}$ $\boxed{7}$:

$$\frac{\begin{array}{l} \overbrace{U\ Ex}^{\boxed{8}}\ \vdash\ \overbrace{U\ M(1:\ \pi)F}^{\boxed{9}}\ +\ \ldots\ +\ U\ Ex\ \vdash\ U\ M(n:\ \pi)F\ + \\ U\ Ex\ \vdash\ \overbrace{U\ M(1,2:\ \pi)F}^{\boxed{10}}\ + U\ Ex\ \vdash\ U\ M(1,3:\ \pi)F\ +\ \ldots\ +\ U\ Ex\ \vdash\ U\ M(n-1,n:\ \pi)F\ + \\ \overbrace{U\ \neg Ex\ \vdash\ T}^{\boxed{11}} \end{array}}{\vdash\ U\ M(\pi)F} \tag{R 27}$$

If $M \in Oper^{E}$ $\boxed{12}$:

$$\frac{\begin{array}{l} U\ Ex\ \vdash\ U\ M(1:\ \pi)F\ ,\ \ldots\ ,\ U\ M(n:\ \pi)F\ , \\ U\ M(1,2:\ \pi)F\ ,\ U\ M(1,3:\ \pi)F\ ,\ \ldots\ ,\ U\ M(n-1,n:\ \pi)F\ + \\ U\ \neg Ex\ \vdash\ T \end{array}}{\vdash\ U\ M(\pi)F} \tag{R 28}$$

Exclusive Control There are two differences between the cases with and without exclusive control. If π_I has exclusive control only those formulas that correspond to the execution of commands from π_I are needed. And exclusive control is lost if execution is impossible.

For

- $I \in [1;n]$
- $\text{First}(\pi_i) \notin \underline{\text{end}}$ for $i \in [1;n]$
- $c := \text{First}(\pi_I)$
- $\text{Label}(c) = \text{EC}$
- $\boxed{13}$ $Ex := Exec^A\big(\text{First}(\pi_I)\big) \vee \bigvee_{\substack{j=1\\ j\neq I}}^{n} Exec^S\big(\text{First}(\pi_I), \text{First}(\pi_j)\big)$

If $M \in Oper^U$:

$$\frac{\begin{array}{l}\overbrace{U\ Ex}^{\boxed{14}} \ \vdash\ U\ M(I:\ \pi)F \ +\ U\ Ex\ \vdash\ U\ M(I,1:\ \pi)F\ +\ \ldots\ +\ U\ Ex\ \vdash\ U\ M(I,I-1:\ \pi)F\ + \\ U\ Ex\ \vdash\ U\ M(I,I+1:\ \pi)F\ +\ \ldots\ +\ U\ Ex\ \vdash\ U\ M(I,n:\ \pi)F\ + \\ \overbrace{U\ \neg Ex\ \vdash\ U\ M(\text{Dropec}(\pi))F}^{\boxed{15}}\end{array}}{\vdash\ U\ M(\pi)F} \quad \text{(R 29)}$$

If $M \in Oper^E$:

$$\frac{\begin{array}{l}U\ Ex\ \vdash\ U\ M(I:\ \pi)F\ ,\ U\ M(I,1:\ \pi)F\ ,\ \ldots\ ,\ U\ M(I,I-1:\ \pi)F\ , \\ U\ M(I,I+1:\ \pi)F\ ,\ \ldots\ ,\ U\ M(I,n:\ \pi)F\ + \\ U\ \neg Ex\ \vdash\ U\ M(\text{Dropec}(\pi))F\end{array}}{\vdash\ U\ M(\pi)F} \quad \text{(R 30)}$$

$\boxed{1}$ The first command of π_i must be executed next because process termination overrules indeterministic scheduling.

$\boxed{2}$ Ex holds precisely if any first command is executable asynchronously or synchronously.

$\boxed{3}$ T is the condition that has to hold if a timeout occurs.

$\boxed{4}$ The conclusion holds because M quantifies universally over the empty set of final states.

$\boxed{5}$ The conclusion does not hold because M quantifies existentially over the empty set of final states.

$\boxed{6}$ F is evaluated one last time in the current state as if the program terminated normally.

$\boxed{7}$ Any processes can be chosen for asynchronous or synchronous execution. In all cases the conclusion needs to hold. If a command is not executable the respective proof goal is closed by the symbolic execution rules.

$\boxed{8}$ There is an executable command.

$\boxed{9}$ This is the case where a command from π_1 is executed asynchronously.

$\boxed{10}$ This is the case where two commands from the processes π_1 and π_2 are executed synchronously.

$\boxed{11}$ If there is no executable command the timeout condition has to hold.

$\boxed{12}$ Any processes can be chosen for asynchronous or synchronous execution. In one of these cases the conclusion needs to hold. If a command is not executable the respective formula is removed by the symbolic execution rules.

$\boxed{13}$ Ex holds precisely if $\text{First}(\pi_I)$ is executable asynchronously or the first of two synchronously executable commands.

$\boxed{14}$ Two cases are distinguished: If Ex holds, the normal scheduling takes place, restricted to commands from π_I, ...

$\boxed{15}$ ... if Ex does not hold, exclusive control is lost.

10.3.4 Unwinding

10.3.4.1 Unwinding with Respect to Asynchronous Execution

Selections For

- $i \in [1;n]$
- $s := \pi_i$
- $c := \mathrm{First}(s) \in \underline{\mathrm{if}} \cup \underline{\mathrm{do}}$
- $r := \mathit{opt}(c)$
- for $j \in [1;r]$: $u_j := \begin{cases} c[j]; \mathrm{Rest}(s) & \text{if } c \in \underline{\text{if}} \\ c[j]; \mathrm{BT}\ c; \mathrm{Rest}(s) & \text{if } c \in \underline{\text{do}} \end{cases}$
- if $\mathit{optelse}(c) = 1$: $u_{r+1} := \begin{cases} c[\mathit{else}]; \mathrm{Rest}(s) & \text{if } c \in \underline{\text{if}} \\ c[\mathit{else}]; \mathrm{BT}\ c; \mathrm{Rest}(s) & \text{if } c \in \underline{\text{do}} \end{cases}$
- $\boxed{1}\ N := \begin{cases} \mathit{true} & \text{if } \mathit{optelse}(c) = 0,\ M \in \mathit{Oper}^U\ \boxed{2} \\ \mathit{false} & \text{if } \mathit{optelse}(c) = 0,\ M \in \mathit{Oper}^E\ \boxed{3} \\ M(i:\ \pi^i(u_{r+1}))F & \text{if } \mathit{optelse}(c) = 1\ \boxed{4} \end{cases}$
- $\boxed{5}\ \mathit{Else} := \bigwedge_{j=1}^{r} \Big(\neg \mathit{Exec}^A\big(\mathrm{First}(u_j)\big) \wedge \bigwedge_{\substack{k=1 \\ k \neq i}}^{n} \neg \mathit{Exec}^S\big(\mathrm{First}(u_j), \mathrm{First}(\pi_k)\big)\Big)$

If $M \in \mathit{Oper}^U$ $\boxed{6}$:

$$\frac{\vdash\ \overbrace{U\ M(i:\ \pi^i(u_1))F}^{\boxed{7}}\ +\ \ldots\ +\ \vdash\ U\ M(i:\ \pi^i(u_r))F\ +\ U\ \mathit{Else}\ \vdash\ \overbrace{U\ N}^{\boxed{8}}}{\vdash\ U\ M(i:\ \pi)F} \tag{R 31}$$

If $M \in \mathit{Oper}^E$ $\boxed{9}$:

$$\frac{\vdash\ U\ M(i:\ \pi^i(u_1))F\ ,\ \ldots\ ,\ U\ M(i:\ \pi^i(u_r))F\ ,\ \overbrace{U\ (\mathit{Else} \wedge N)}^{\boxed{10}}}{\vdash\ U\ M(i:\ \pi)F} \tag{R 32}$$

Atomic Sequences For

- $i \in [1;n]$
- $S := \pi_i$
- $C := \mathrm{First}(S) \in \underline{\mathrm{atomic}}$
- $s := C[\mathit{body}]$
- $c := \mathrm{First}(s)$

and

- $c \in \underline{\mathrm{if}} \cup \underline{\mathrm{do}}$: $\boxed{11}$
  Let
  - r and u_j for $j \in [1; r + \mathit{optelse}(c)]$ be defined as above,
  - $U_j := \mathrm{ATOMIC}\{u_j\}; \mathrm{Rest}(S)$ for $j \in [1; r + \mathit{optelse}(c)]$,
  - Else and N be as above, but with u_{r+1} replaced with U_{r+1}.

  If $M \in \mathit{Oper}^U$:

$$\frac{\vdash\ U\ M(i:\ \pi^i(U_1))F\ +\ \ldots\ +\ \vdash\ U\ M(i:\ \pi^i(U_r))F\ +\ U\ \mathit{Else}\ \vdash\ U\ N}{\vdash\ U\ M(i:\ \pi)F} \tag{R 33}$$

  If $M \in \mathit{Oper}^E$:

$$\frac{\vdash\ U\ M(i:\ \pi^i(U_1))F\ ,\ \ldots\ ,\ U\ M(i:\ \pi^i(U_r))F\ ,\ U\ (\mathit{Else} \wedge N)}{\vdash\ U\ M(i:\ \pi)F} \tag{R 34}$$

- $c \in \underline{\text{elementary}} \cup \underline{\text{atomic}}$ $\boxed{12}$

  Let $S' := c; \text{ECAtomic}(\text{Rest}(s)); \text{Rest}(S)$, then:

$$\frac{\vdash\ U\ M(i:\ \pi^i(S'))F}{\vdash\ U\ M(i:\ \pi)F} \tag{R 35}$$

$\boxed{1}$ N is used to treat the cases with and without else option together: If c has no else option N can be eliminated immediately by other rules. In an implementation separate rules can be used instead.

$\boxed{2}$ In this case the proof goal with N can be immediately closed.

$\boxed{3}$ In this case the formula with N cannot be derived. This is sound, but inefficient, because the formula *Else* cannot be removed.

$\boxed{4}$ In this case the else option is unwound.

$\boxed{5}$ *Else* holds precisely if $\text{First}(u_j)$ is not executable asynchronously or as the first part of a synchronous execution of two commands for any $j \in [1; r]$, i. e. if a possible else option needs to be unwound.

$\boxed{6}$ In all cases of the selection the conclusion needs to hold.

$\boxed{7}$ The first option of c is unwound.

$\boxed{8}$ If no non-else option is executable a possible else option is unwound.

$\boxed{9}$ In one case of the selection the conclusion needs to hold.

$\boxed{10}$ If no non-else option is executable a possible else option is unwound.

$\boxed{11}$ These rules are similar to the rules (R 31) and (R 32). They operate on s and $c = \text{First}(s)$, too, but the difference is that here s is the body of the atomic sequence and not the whole process. For $j \in [1; r + \mathit{optelse}(c)]$, U_j is defined such that the results of unwinding s, the command sequences u_j, are properly integrated into π_i.

$\boxed{12}$ This rule pulls the first command, which may not be a selection, out of the atomic sequence as in the definition of Unwind^{A}.

10.3.4.2 Unwinding with Respect to Synchronous Execution

These rules are essentially the same as for asynchronous execution. The differences correspond to those between the mappings Unwind^{A} and Unwind^{S}. The variable $k \in \{i, j\}$ is used to treat the independent, but symmetric cases of unwinding the first part, $k = i$, and of unwinding the second part, $k = j$, together.

Then formally, the only differences between the asynchronous and the synchronous case are that the applicability condition $k \in \{i, j\}$ is added, the tag i is replaced with i, j and that all other occurrences of i are replaced with k. Also the else options of selections can be ignored in the synchronous case.

Nevertheless the rules are still given explicitly below for clarity and completeness.

Selections For

- $i, j \in [1; n]$
- $i \neq j$
- $k \in \{i, j\}$
- $s := \pi_k$
- $c := \text{First}(s) \in \underline{\text{if}} \cup \underline{\text{do}}$
- $r := \mathit{opt}(c)$
- for $l \in [1; r]$, u_l is defined as in the asynchronous case

If $M \in \mathit{Oper}^U$:

$$\frac{\vdash\ U\ M(i,j:\ \pi^k(u_1))F\ +\ \dots\ +\ \vdash\ U\ M(i,j:\ \pi^k(u_r))F}{\vdash\ U\ M(i,j:\ \pi)F} \tag{R 36}$$

If $M \in \mathit{Oper}^E$:

$$\frac{\vdash\ U\ M(i,j:\ \pi^k(u_1))F\ ,\ \dots\ ,\ U\ M(i,j:\ \pi^k(u_r))F}{\vdash\ U\ M(i,j:\ \pi)F} \tag{R 37}$$

Atomic Sequences For

- $i, j \in [1; n]$
- $i \neq j$
- $k \in \{i, j\}$
- $S := \pi_k$
- $C := \text{First}(S) \in \underline{\text{atomic}}$
- $s := C[\mathit{body}]$
- $c := \text{First}(s)$

and

- $c \in \underline{\text{if}} \cup \underline{\text{do}}$
  Let r and U_l for $l \in [1; r]$ be defined as in the asynchronous case:
  - If $M \in \mathit{Oper}^U$:

$$\frac{\vdash\ U\ M(i,j:\ \pi^k(U_1))F\ +\ \ldots\ +\ \vdash\ U\ M(i,j:\ \pi^k(U_r))F}{\vdash\ U\ M(i,j:\ \pi)F} \tag{R 38}$$

  - If $M \in \mathit{Oper}^E$:

$$\frac{\vdash\ U\ M(i,j:\ \pi^k(U_1))F\ ,\ \ldots\ ,\ U\ M(i,j:\ \pi^k(U_r))F}{\vdash\ U\ M(i,j:\ \pi)F} \tag{R 39}$$

- $c \in \underline{\text{elementary}} \cup \underline{\text{atomic}}$. Let S' be defined as in the asynchronous case:

$$\frac{\vdash\ U\ M(i,j:\ \pi^k(S'))F}{\vdash\ U\ M(i,j:\ \pi)F} \tag{R 40}$$

10.3.5 Symbolic Execution

There are twelve symbolic execution rules: for six modal operators and two types of execution (asynchronous or synchronous).

There are three symmetries between the rules that are used to present them in a compact notation:

- Firstly, for the two sets Oper^U and Oper^E of modal operators and the two types of execution four rule schemes are given. These govern the construction of the tree of traces.
- Then the twelve rules are defined relative to the four rule schemes. This process primarily depends on which of the sets Oper_U, Oper_E and Oper_F the modal operator belongs to. This part of the rule definition governs how the truth values along the traces determine the truth value of the modality.
- Finally, the rules are given another time using six rule schemes, one for each modal operator and abstracted from the type of execution. In this representation the structure of the rules is most easily visible.

10.3.5.1 Rule schemes

Rule Schemes for Asynchronous Execution For

- $i \in [1; n]$
- $c := \text{First}(\pi_i) \in \underline{\text{elementary}}$ $\boxed{1}$

If $M \in \mathit{Oper}^U$:

$$\frac{\overbrace{U\ \mathit{Exec}^A(c)}^{\boxed{2}} \ \vdash\ \overbrace{U \cdot \mathit{Eff}^A(c)}^{\boxed{3}}\ \overbrace{M(\boxed{4}\,\mathrm{rem}^A(\pi, i))F}^{\boxed{5}}}{\vdash\ U\ M(i:\ \pi)F} \tag{R 41}$$

If $M \in \mathit{Oper}^E$:

$$\frac{U\ \mathit{Exec}^A(c)\ \vdash\ U \cdot \mathit{Eff}^A(c)\ M(\mathrm{rem}^A(\pi, i))F\ +\ \overbrace{U\ \neg \mathit{Exec}^A(c)\ \vdash}^{\boxed{6}}}{\vdash\ U\ M(i:\ \pi)F} \tag{R 42}$$

Rule Schemes for Synchronous Execution These rule schemes are essentially the same as for asynchronous execution. The differences are that the tag in the proof goal, the executability check, the effect and the remaining program are replaced by the corresponding objects for the synchronous case. For

- $i, j \in [1; n]$
- $i \neq j$
- $c := \mathrm{First}(\pi_i) \in \underline{\text{elementary}}$
- $d := \mathrm{First}(\pi_j) \in \underline{\text{elementary}}$

If $M \in \mathit{Oper}^U$:

$$\frac{U\ \mathit{Exec}^S(c, d)\ \vdash\ U \cdot \mathit{Eff}^S(c, d)\ M(\mathrm{rem}^S(\pi, i, j))F}{\vdash\ U\ M(i, j:\ \pi)F} \tag{R 43}$$

If $M \in \mathit{Oper}^E$:

$$\frac{U\ \mathit{Exec}^S(c, d)\ \vdash\ U \cdot \mathit{Eff}^S(c, d)\ M(\mathrm{rem}^S(\pi, i, j))F\ +\ U\ \neg \mathit{Exec}^S(c, d)\ \vdash}{\vdash\ U\ M(i, j:\ \pi)F} \tag{R 44}$$

$\boxed{1}$ This condition ensures that the unwinding rules are not applicable anymore.

$\boxed{2}$ If c is asynchronously executable ...

$\boxed{3}$ ... after applying its effect ...

$\boxed{4}$ ... and removing the tag ...

$\boxed{5}$ ... the conclusion with the remaining program after execution of c must hold.

$\boxed{6}$ This sequent is the only difference between the two rules. It should be kept in mind that this rule is given without the context: If c is not executable the conclusion is false and can be removed and one of the other formulas in the implicit context must be proven. In the same case for $M \in \mathit{Oper}^U$ no such premise is needed because the conclusion is true.

Remark The formulas $\mathit{Exec}^A(c)$ and $\mathit{Exec}^S(c, d)$ imply the formulas Ex, which is introduced by the scheduling rules, and possibly $\neg \mathit{Else}$, which is introduced by the unwinding rules. This can be used in an implementation of the calculus to increase the efficiency (see section 13.4).

10.3.5.2 Definition of the Rules Relative to the Rule Schemes

The rules for the modal operators can be given relative to the appropriate rule scheme, depending only on the type of the modal operator and the type of execution.

- For $M \in \mathit{Oper}_F$: The rule scheme is unchanged. For the conclusion holds if F holds in the final state, i. e. when the modal operator is empty.

- For $M \in Oper_E$: The formula $U\ F$ is added to the succedent of the first premise of the rule scheme. For the states within the trace are quantified existentially and therefore the conclusion holds whenever F holds during the execution.
- For $M \in Oper_U$: The sequents $U\ Exec^A(c)\ \vdash\ U\ F$ or $U\ Exec^S(c,d)\ \vdash\ U\ F$, respectively, are added to the premises of the rule scheme. For the states within the trace are quantified universally and therefore the conclusion holds if F always holds during the execution.

10.3.5.3 Abstraction from the Type of Execution

The symbolic execution rules are given another time below. The notation includes the context, and distinguishes between the modal operators. But the formulas that occur in the rules are abbreviated as follows. These abbreviations allow to treat the asynchronous and the synchronous case together.

	asynchronous case	synchronous case
$E =$	$U\ Exec^A(c)$	$U\ Exec^S(c,d)$
$\neg E =$	$U\ \neg Exec^A(c)$	$U\ \neg Exec^S(c,d)$
$C =$	$U\ M(i:\ \pi)F$	$U\ M(i,j:\ \pi)F$
$C' =$	$U \cdot Eff^A(c)\ M(\mathrm{rem}^A(\pi,i))F$	$U \cdot Eff^S(c,d)\ M(\mathrm{rem}^S(\pi,i,j))F$
$A =$	$U\ F$	$U\ F$

With these abbreviations the rules are:

	$M \in Oper^U$	$M \in Oper^E$
$M \in Oper_F$	$M = []$: $\dfrac{\Gamma, E\ \vdash\ C', \Delta}{\Gamma\ \vdash\ C, \Delta}$	$M = \langle\rangle$: $\dfrac{\Gamma, E\ \vdash\ C', \Delta\ +\ \Gamma, \neg E\ \vdash\ \Delta}{\Gamma\ \vdash\ C, \Delta}$
$M \in Oper_U$	$M = [[]]$: $\dfrac{\Gamma, E\ \vdash\ A, \Delta\ +\ \Gamma, E\ \vdash\ C', \Delta}{\Gamma\ \vdash\ C, \Delta}$	$M = \langle[]\rangle$: $\dfrac{\Gamma, E\ \vdash\ A, \Delta\ +\ \Gamma, E\ \vdash\ C', \Delta\ +\ \Gamma, \neg E\ \vdash\ \Delta}{\Gamma\ \vdash\ C, \Delta}$
$M \in Oper_E$	$M = [\langle\rangle]$: $\dfrac{\Gamma, E\ \vdash\ C', A, \Delta}{\Gamma\ \vdash\ C, \Delta}$	$M = \langle\langle\rangle\rangle$: $\dfrac{\Gamma, E\ \vdash\ C', A, \Delta\ +\ \Gamma, \neg E\ \vdash\ \Delta}{\Gamma\ \vdash\ C, \Delta}$

10.3.6 Gödelization

The following rules[50] comprise the indirect approach to eliminate programs, which is based on Gödelization. They are needed for the completeness of **C**, but will not be present in an implementation. All other rules for the elimination of programs are not needed for completeness, but are included because they can be implemented efficiently.

For

- $M \in Oper$,
- $\pi \in Prog$
- $F, G \in Form$ such that G is first-order,
- $\phi^{M,\pi,G}$ denoting the formula constructed in Theorem 2, which is effectively computable from M, π and G:

$$\frac{\vdash\ F'}{\vdash\ F} \tag{R 45}$$

$$\frac{F'\ \vdash}{F\ \vdash}$$

where F' arises from F by replacing some occurrence of $M(\pi)G$ in F with $\phi^{M,\pi,G}$.

[50]For simplicity the two rules are referenced with the same number.

10.4 Additional Rules for Modal Operators

The following rules are not needed, but they are included to increase efficiency.

Distributivity For formulas F and G, a program π and

- $M \in \{[], [[]]\}$:

$$\frac{\vdash\ M(\pi)F \wedge M(\pi)G}{\vdash\ M(\pi)(F \wedge G)} \tag{R 46}$$

- $M \in \{\langle\rangle, \langle\langle\rangle\rangle\}$:

$$\frac{\vdash\ M(\pi)F \vee M(\pi)G}{\vdash\ M(\pi)(F \vee G)} \tag{R 47}$$

These two rules do not necessarily increase efficiency, because π must be executed twice. It can even be useful to include further rules that apply these rules in the opposite direction.

Generalization This rule is stated with context, i. e. $\Gamma = \Delta = \emptyset$. For a formula F, a program π and a modal operator $M \neq \langle\rangle$:

$$\frac{\vdash\ F}{\vdash\ M(\pi)F} \tag{R 48}$$

The context must have been eliminated by the weakening rule before this rule may be applied. That is necessary because otherwise, even without free variables, the rule is not sound. A simple counter-example is

$$\frac{\vdash\ Y_1^I =^I c_0\ ,\ \neg Y_1^I =^I c_0}{\vdash\ Y_1^I =^I c_0\ ,\ [[Y_1^I := c_0]]\neg Y_1^I =^I c_0}$$

10.5 Updates

Updates and First-Order Logic For an update u, $\diamond \in \{\wedge, \vee\}$ and formulas F and G:

$$\frac{\vdash\ u\,F \diamond u\,G}{\vdash\ u\,(F \diamond G)} \tag{R 49}$$

$$\frac{u\,F \diamond u\,G\ \vdash}{u\,(F \diamond G)\ \vdash} \tag{R 50}$$

$$\frac{\vdash\ \neg u\,F}{\vdash\ u\,\neg F} \tag{R 51}$$

$$\frac{\neg u\,F\ \vdash}{u\,\neg F\ \vdash} \tag{R 52}$$

For an update u, $Q \in \{\forall, \exists\}$, $x \in LV$ such that x does not occur in u, and a formula F:

$$\frac{\vdash\ Qx\,u\,F}{\vdash\ u\,Qx\,F} \tag{R 53}$$

$$\frac{Qx\,u\,F\ \vdash}{u\,Qx\,F\ \vdash} \tag{R 54}$$

Application of Updates The following rules[51] eliminate updates by applying the corresponding update substitution. For formulas F, G and an update u such that G is first-order and no non-primary non-rigid functions symbols occur in G:

$$\frac{\vdash\ F'}{\vdash\ F} \qquad \text{(R 55)}$$

$$\frac{F'\ \vdash}{F\ \vdash}$$

where F' arises from F by replacing some occurrence of the formula $u\, G$ in F with $\sigma_u(G)$.

Indirect Elimination of Updates This rule is stated with context. For $\Gamma, \Delta \subseteq \mathit{Form}$, an update $u = (f, (t_1, \ldots, t_n), t)$ where $t \in \mathit{Terms}(S)$ and a formula F:

$$\frac{\Gamma\ \vdash\ u\, F\ ,\ \Delta}{\Gamma\ \vdash\ f(t_1, \ldots, t_n) =^S t \rightarrow F\ ,\ \Delta} \qquad \text{(R 56)}$$

This rule can be used to eliminate updates if a program does not terminate, which is helpful in proofs that use the induction rule.

Without restriction of the applicability conditions the global soundness of this rule cannot be strong. A simple counter-example is given by:

$$\frac{\mathit{proc}(c_0) =^P \mathit{fin}\ \vdash\ (\mathit{proc}, c_0, \mathit{run})\ \mathit{proc}(c_0) =^P \mathit{fin}}{\mathit{proc}(c_0) =^P \mathit{fin}\ \vdash\ \mathit{proc}(c_0) =^P \mathit{run} \rightarrow \mathit{proc}(c_0) =^P \mathit{fin}}$$

Update Generalization This rule is stated with context, i. e. $\Gamma = \Delta = \emptyset$. For a formula F and an update u:

$$\frac{\vdash\ F}{\vdash\ u\, F} \qquad \text{(R 57)}$$

The rule is not needed, but simplifies some proofs.

See the remark after rule (R 48) why no context may be present.

10.6 Function and Predicate Symbols

10.6.1 Abbreviations

Several abbreviations are introduced in order to state the axioms for function and predicate symbols in a readable form:

- The word x_i^S is abbreviated by x_i for $S \in \Sigma$ and $i \in \mathbb{N}^*$. The symbols $=^S$ and $\leq^S$ are abbreviated by $=$ and $\leq$, respectively.

  Intuitively, the sort symbols as superscripts of logical variables and the predicate symbols $=^S$ and $\leq^S$ are left out.

  The superscripts of function symbols and variable occurrences behind a quantifier are not left out. Therefore, using the signature definitions, it is always possible to determine the sort of any term in a formula that uses these abbreviations.

- For $S \in \Sigma_P$, $t_1, t_2, t_3 \in \mathit{Terms}(S)$ the word $(t_1 \leq t_2 \wedge \neg t_1 = t_2)$ is abbreviated by $t_1 < t_2$, and for $\circ, \bullet \in \{=, \leq, <\}$ the word $(t_1 \circ t_2 \wedge t_2 \bullet t_3)$ is abbreviated by $t_1 \circ t_2 \bullet t_3$.

These abbreviations can be used simultaneously without any ambiguities.

[51] For simplicity the two rules are referenced with the same number.

10.6.2 Oracle

The standard structure allows to define the natural numbers. Therefore an oracle rule scheme is needed the set of instances of which is not recursively enumerable.

For all sequents $\Gamma \vdash \Delta$ that only contain first-order formulas[52] and that satisfy $\overline{\text{val}_g}(\Gamma \vdash \Delta) = 1$ for all $g \in \mathbf{G}$:

$$\frac{}{\Gamma \vdash \Delta} \tag{R 58}$$

All the remaining axioms in this section could be derived from the oracle rule. They are included because the oracle rule will not be present in an implementation of **C**.

10.6.3 Non-rigid Function Symbols

Primary Symbols The following rules require that the primary non-rigid function symbols are defined for only finitely many arguments. For $S \in \Sigma_P$:

$$\frac{}{\vdash \exists x_1^I \exists x_2^I \forall x_3^I \forall x_4^I \big((x_1 \leq x_3 \vee x_2 \leq x_4) \rightarrow \mathit{var}^S(x_3, x_4) = \bot^S\big)} \tag{R 59}$$

$$\frac{}{\vdash \exists x_1^I \exists x_2^I \; \forall x_3^I \forall x_4^I \big((x_3 \leq x_1 \vee x_4 \leq x_2) \rightarrow \mathit{var}^S(x_3, x_4) = \bot^S\big)} \tag{R 60}$$

$$\frac{}{\vdash \exists x_1^C \forall x_2^C (x_1 \leq x_2 \rightarrow \mathit{cont}(x_2) = \bot^Q)} \tag{R 61}$$

$$\frac{}{\vdash \exists x_1^I \forall x_2^I (x_1 \leq x_2 \rightarrow \mathit{proc}(x_2) = \bot^P)} \tag{R 62}$$

$$\frac{}{\vdash \exists x_1^I \forall x_2^I (x_2 \leq x_1 \rightarrow \mathit{proc}(x_2) = \bot^P)} \tag{R 63}$$

Definable Symbols The non-primary non-rigid function symbols can be defined from the remaining symbols. The following rules[53] allow to eliminate occurrences of these definable symbols outside programs:

$$\frac{\vdash F'}{\vdash F} \tag{R 64}$$

$$\frac{F' \vdash}{F \vdash}$$

where F' arises from F by applying one of the following term replacing rules[54] to some occurrence in F outside a program:

- replace $\mathit{lengthc}(c)$ with $\mathit{length}(\mathit{cont}(c))$ for $c \in \mathit{Terms}(C)$
- replace $\mathit{pollfirst}^{\sigma,S}(c, t_1, \ldots, t_r)$ with

  $$[\mathit{match}^{\sigma,S}(\mathit{read}^S(\mathit{cont}(c), c_1), t_1, \ldots, t_r) \; ? \; c_1 \; : \; c_0]$$

  or $\mathit{pollany}^{\sigma,S}(c, t_1, \ldots, t_r)$ with

  $$[\exists x_1^I \mathit{match}^{\sigma,S}(\mathit{read}^S(\mathit{cont}(c), x_1), t_1, \ldots, t_r) \; ? \; c_1 \; : \; c_0]$$

  for $S \in \Sigma_P$, $\sigma = \{i_1, \ldots, i_r\} \subseteq [1; \text{Length}(S)]$, $c \in \mathit{Terms}(C)$, $t_j \in \mathit{Terms}(S_{i_j})$ for $j \in [1; r]$:

These rules require that $\mathit{pollfirst}^{\sigma,S}$ has the value c_1 if the first element in the queue $\mathit{cont}(c)$ matches the pattern $(t_1, \ldots, t_r)$ and c_0 otherwise. $\mathit{pollany}^{\sigma,S}$ has the value c_1 if any element in the queue matches the pattern and c_0 otherwise.

[52]Non-rigid function symbols may occur in these formulas.

[53]For simplicity the two rules are referenced with the same number.

[54]The rules must be stated in this form because equality axioms would not allow to replace terms that occur within the scope of a modal operator. The latter is not needed for the completeness proof, but makes it simpler.

10.6.4 Rigid Function Symbols and Universes

10.6.4.1 Universes

For $S \in \Sigma_P \setminus \Sigma_E$:

$$\frac{}{\vdash\ \forall x_{n+1}^{S} \exists x_1^{S_1} \dots \exists x_n^{S_n}\ x_{n+1} = \mathit{tuple}^S(x_1, \dots, x_n)} \tag{R 65}$$

$$\frac{}{\vdash\ \forall x_1^{P}(x_1 = \mathit{run} \vee x_1 = \mathit{fin} \vee x_1 = \bot^{P})} \tag{R 66}$$

10.6.4.2 Definedness

For all rigid function symbols f such that

$$f \notin \{/\} \cup \{\bot^S | S \in \Sigma\} \cup \{\mathit{read}^S, \mathit{sendpos}^S | S \in \Sigma_P\} \cup \left\{\mathit{recpos}^{\sigma,S} | S \in \Sigma_P, \sigma \subseteq [1; \mathrm{Length}(S)]\right\}$$

and $\mathrm{Sign}(f) = S^1\ \dots\ S^n\ S$ for $n \in \mathbb{N}$, and for $t_i \in \mathit{Terms}(S^i)$ for $i \in [1; n]$:

$$\frac{}{t_1 = \bot^{S^1} \vee \dots \vee t_n = \bot^{S^n}\ \vdash\ f(t_1, \dots, t_n) = \bot^S} \tag{R 67}$$

$$\frac{}{f(t_1, \dots, t_n) = \bot^S\ \vdash\ t_1 = \bot^{S^1}, \dots, t_n = \bot^{S^n}} \tag{R 68}$$

The first rule expresses that functions are undefined if arguments are undefined. The second rule expresses the opposite direction. In the case $n = 0$ these rules state that constant symbols are defined. The excluded function symbols can be undefined even if all arguments are defined; the corresponding rules are given in the specific sections below.

10.6.4.3 Initial Values

$$\frac{}{\vdash\ \mathit{ini}^I = c_0} \tag{R 69}$$

$$\frac{}{\vdash\ \mathit{ini}^C = \mathit{nil}^C} \tag{R 70}$$

For $S \in \Sigma_P \setminus \Sigma_E$:

$$\frac{}{\vdash\ \mathit{ini}^S = \mathit{tuple}^S(\mathit{ini}^{S_1}, \dots, \mathit{ini}^{S_n})} \tag{R 71}$$

10.6.4.4 Integers

The following rules correspond to the usual axioms for the integers for the vocabulary $(0, 1, +, -, \cdot)$. For $i, j, k \in \mathit{Terms}(I)$:

Addition and Subtraction

$$\frac{}{\vdash\ i + (j + k) = (i + j) + k} \tag{R 72}$$

$$\frac{}{\vdash\ i + c_0 = i} \tag{R 73}$$

$$\frac{}{\neg i = \bot^I\ \vdash\ i + (c_0 - i) = c_0} \tag{R 74}$$

$$\frac{}{\vdash\ i + j = j + i} \tag{R 75}$$

$$\frac{}{\vdash\ i - j = i + (c_0 - j)} \tag{R 76}$$

Multiplication

$$\frac{}{\vdash\ (i \cdot j) \cdot k = i \cdot (j \cdot k)} \tag{R 77}$$

$$\frac{}{\vdash\ i \cdot c_1 = i} \tag{R 78}$$

$$\frac{}{\vdash\ i \cdot j = j \cdot i} \tag{R 79}$$

$$\frac{}{\vdash\ i \cdot (j + k) = (i \cdot j) + (i \cdot k)} \tag{R 80}$$

Division For $i, j, k \in Terms(I)$:

$$\frac{}{\neg j = \bot^I\ ,\ \neg j = c_0\ ,\ i = c_0\ \vdash\ i/j = c_0} \tag{R 81}$$

$$\frac{c_0 < i\ ,\ c_0 < j\ \vdash\ k \cdot j \leq i < (k + c_1) \cdot j}{c_0 < i\ ,\ c_0 < j\ \vdash\ i/j = k} \tag{R 82}$$

$$\frac{c_0 < i\ ,\ j < c_0\ \vdash\ k \cdot j \leq i < (k - c_1) \cdot j}{c_0 < i\ ,\ j < c_0\ \vdash\ i/j = k} \tag{R 83}$$

$$\frac{i < c_0\ ,\ c_0 < j\ \vdash\ (k - c_1) \cdot j < i \leq k \cdot j}{i < c_0\ ,\ c_0 < j\ \vdash\ i/j = k} \tag{R 84}$$

$$\frac{i < c_0\ ,\ j < c_0\ \vdash\ (k + c_1) \cdot j < i \leq k \cdot j}{i < c_0\ ,\ j < c_0\ \vdash\ i/j = k} \tag{R 85}$$

The following rules express the definedness condition.

$$\frac{}{i = \bot^I \vee j = \bot^I \vee j = c_0\ \vdash\ i/j = \bot^I} \tag{R 86}$$

$$\frac{}{i/j = \bot^I\ \vdash\ i = \bot^I\ ,\ j = \bot^I\ ,\ j = c_0} \tag{R 87}$$

Induction The induction rule includes more cases than the classical rule: for the negative numbers and for the undefined element; the additional formulas $c_0 \leq^I x$ and $x \leq^I c_0$ in the antecedents are included for efficiency. For all formulas F with $FV(F) = \{x\} \subseteq LV(I)$:

$$\frac{\vdash\ \{c_0/x\}F\ +\ c_0 \leq^I x\ ,\ F\ \vdash\ \{(x + c_1)/x\}F\ +\ x \leq^I c_0\ ,\ F\ \vdash\ \{(x - c_1)/x\}F\ +\ \ \vdash\ \{\bot^I/x\}F}{\vdash\ F} \tag{R 88}$$

Constant Symbols The following rules are needed to define the constant symbols for the integers. It may be difficult to include these rules in an implementation. In that case the symbols c_n for $n \in \mathbb{Z} \setminus \{0, 1\}$ must be considered as abbreviations and not as a part of the language. For $n \in \mathbb{N}^*$:

$$\frac{}{\vdash\ c_n = \underbrace{c_1 + \ldots + c_1}_{n \text{ repetitions}}} \tag{R 89}$$

$$\frac{}{\vdash\ c_{-n} = c_0 \underbrace{- c_1 - \ldots - c_1}_{n \text{ repetitions}}} \tag{R 90}$$

10.6.4.5 Channels

These axioms correspond to the usual axioms for the natural numbers with zero symbol nil^C and successor function symbol *incchan*.

For all $c, d \in Terms(C)$:

$$\frac{}{incchan(c) = nil^C \ \vdash} \tag{R 91}$$

$$\frac{}{incchan(c) = incchan(d) \ \vdash \ c = d} \tag{R 92}$$

For all formulas F with $FV(F) = \{x\} \subseteq LV(C)$ (Induction):

$$\frac{\vdash \ \{nil^C/x\}F \ + \ F \ \vdash \ \{nextchan(x)/x\}F \ + \ \vdash \ \{\bot^C/x\}F}{\vdash \ F} \tag{R 93}$$

10.6.4.6 Composed Sorts

For $S \in \Sigma_P \setminus \Sigma_E$ with Length$(S) = n$, $t_i \in Terms(S_i)$ for $i \in [1; n]$ and $t \in Terms(S)$:

$$\frac{}{\neg t_1 = \bot^{S_1}, \ldots, \neg t_n = \bot^{S_n} \ \vdash \ elem_i^S(tuple^S(t_1, \ldots, t_n)) = t_i} \tag{R 94}$$

$$\frac{}{\vdash \ tuple^S(elem_1^S(t), \ldots, elem_n^S(t)) = t} \tag{R 95}$$

For $S \in \Sigma_E$ and $t \in Terms(S)$:

$$\frac{}{\vdash \ elem_1^S(t) = t} \tag{R 96}$$

$$\frac{}{\vdash \ tuple^S(t) = t} \tag{R 97}$$

10.6.4.7 Queues

For all $m \in \mathbb{N}$, $S \in \Sigma$, $q \in Terms(Q)$, $a, b \in Terms(S)$, $i, j \in Terms(I)$ and with the following abbreviations

- $I_1(S, q, a, i) = \neg q = \bot^Q \wedge \neg a = \bot^S \wedge \neg i = \bot^I$
  which intuitively means that $insert^S(q, a, i)$ is defined
- $I_2(S, q, a, i) = sor^S(q) \wedge c_1 \leq i \leq length(q) + c_1 \wedge length(q) < cap(q)$
  which intuitively means that $insert^S(q, a, i)$, if defined, is different from q
- $I(S, q, a, i) = I_1(S, q, a, i) \wedge I_2(S, q, a, i)$
- $R_1(q, i) = \neg q = \bot^Q \wedge \neg i = \bot^I$
  which intuitively means that $R(q, i)$ is defined
- $R_2(q, i) = c_1 \leq i \leq length(q)$
  which intutively means that $R(q, i)$, if defined, is different from q
- $R(q, i) = R_1(q, i) \wedge R_2(q, i)$

the axioms for queues are as follows.

There are three types of defined terms of the sort Q: The empty queues (of the form $()^{m,S}$), insertion terms (of the form $insert^S(q, a, i)$) and removal terms (of the form $remove(q, i)$). The following axioms determine the action of the rigid function symbols that take arguments of the sort Q for empty queues and insertion terms. Several axioms are given for clarity although they are not needed, for example the axioms for the length and capacity of removal terms. When reducing terms to canonical forms in Lemma 17 the equalities will be used to replace the terms on the left side with the terms on the right side.

Length The following axioms define the length of a queue.

$$\frac{}{\vdash\ \mathit{length}(()^{m,S}) = c_0} \tag{R 98}$$

$$\frac{}{I(S,q,a,i)\ \vdash\ \mathit{length}(\mathit{insert}^S(q,a,i)) = \mathit{length}(q) + c_1} \tag{R 99}$$

$$\frac{}{R(q,i)\ \vdash\ \mathit{length}(\mathit{remove}(q,i)) = \mathit{length}(q) - c_1} \tag{R 100}$$

Capacity The following axioms[55] define the capacity of a queue.

$$\frac{}{\vdash\ \mathit{cap}(()^{m,S}) = c_m} \tag{R 101}$$

$$\frac{}{I_1(S,q,a,i)\ \vdash\ \mathit{cap}(\mathit{insert}^S(q,a,i)) = \mathit{cap}(q)} \tag{R 102}$$

$$\frac{}{R_1(q,i)\ \vdash\ \mathit{cap}(\mathit{remove}(q,i)) = \mathit{cap}(q)} \tag{R 103}$$

Removal This axiom defines that queues are unchanged by removals in certain cases (which includes the case that q is an empty queue).

$$\frac{}{R_1(q,i), \neg R_2(q,i)\ \vdash\ \mathit{remove}(q,i) = q} \tag{R 104}$$

The following axioms define the removal of elements from queues.

$$\frac{}{I(S,q,a,i), j \le c_0\ \vdash\ \mathit{remove}(\mathit{insert}^S(q,a,i),j) = \mathit{insert}^S(q,a,i)} \tag{R 105}$$

$$\frac{}{I(S,q,a,i), c_1 \le j < i\ \vdash\ \mathit{remove}(\mathit{insert}^S(q,a,i),j) = \mathit{insert}^S(\mathit{remove}(q,j),a,i-c_1)} \tag{R 106}$$

$$\frac{}{I(S,q,a,i), i = j\ \vdash\ \mathit{remove}(\mathit{insert}^S(q,a,i),j) = q} \tag{R 107}$$

$$\frac{}{I(S,q,a,i), i < j\ \vdash\ \mathit{remove}(\mathit{insert}^S(q,a,i),j) = \mathit{insert}^S(\mathit{remove}(q,j-c_1),a,i)} \tag{R 108}$$

Insertion This axiom states that queues are unchanged by insertions in certain cases.

$$\frac{}{I_1(S,q,a,i), \neg I_2(S,q,a,i)\ \vdash\ \mathit{insert}^S(q,a,i) = q} \tag{R 109}$$

This axiom changes the order of different insertions which is needed for the reduction to a canonical form.

$$\frac{}{I(S,q,a,i), i < j\ \vdash\ \mathit{insert}^S(\mathit{insert}^S(q,a,i),b,j) = \mathit{insert}^S(\mathit{insert}^S(q,b,j-c_1),a,i)} \tag{R 110}$$

The rules for the case that the sort superscripts of the function symbols are different can be derived.

Reading These axioms define how elements are read from a queue.

$$\frac{}{I(S,q,a,i), j < i\ \vdash\ \mathit{read}^S(\mathit{insert}^S(q,a,i),j) = \mathit{read}^S(q,j)} \tag{R 111}$$

$$\frac{}{I(S,q,a,i), j = i\ \vdash\ \mathit{read}^S(\mathit{insert}^S(q,a,i),j) = a} \tag{R 112}$$

$$\frac{}{I(S,q,a,i), i < j\ \vdash\ \mathit{read}^S(\mathit{insert}^S(q,a,i),j) = \mathit{read}^S(q,j-c_1)} \tag{R 113}$$

[55]The first rule may be difficult to implement, but for every sequent the greatest value of m for which the rule may be needed can be computed. Then this rule scheme can be replaced with finitely many instances.

$$\frac{}{R(q,i), i \le j \;\vdash\; read^S(remove(q,i),j) = read^S(q, j + c_1)} \quad \text{(R 114)}$$

$$\frac{}{R(q,i), j < i \;\vdash\; read^S(remove(q,i),j) = read^S(q, j)} \quad \text{(R 115)}$$

These rules express the definedness condition.

$$\frac{}{q = \bot^Q \lor i = \bot^I \lor \neg sor^S(q) \lor \neg c_1 \le i \le length(q) \;\vdash\; read^S(q,i) = \bot^S} \quad \text{(R 116)}$$

$$\frac{}{read^S(q,i) = \bot^S \;\vdash\; q = \bot^Q \;,\; i = \bot^I \;,\; \neg sor^S(q) \;,\; \neg c_1 \le i \le length(q)} \quad \text{(R 117)}$$

The rules for the case that the sort superscripts of the function symbols are different can be derived.

Send Position for Sorted Sending In the first rule, intuitively, F means that all elements in q up to and excluding the position i are smaller than t. G means that the element at position i is equal or greater than t or, if no such element exists, that i is one greater than the length of q. Those are precisely the conditions for $sendpos^S(q,t)$ to have the defined value i.

$$\frac{\neg q = \bot^Q \;,\; \neg t = \bot^S \;,\; \neg i = \bot^I \;,\; sor^S(q) \;\vdash\; F \land G}{\neg q = \bot^Q \;,\; \neg t = \bot^S \;,\; \neg i = \bot^I \;,\; sor^S(q) \;\vdash\; sendpos^S(q,t) = i} \quad \text{(R 118)}$$

where

$$F = \forall x_1^I \big((c_1 \le x_1 < i) \to read^S(q, x_1) < t\big)$$

$$G = (t \le read^S(q,i) \lor i = length(q) + c_1)$$

The following rules express the definedness condition. In particular if the sort of q and t do not match the value of $sendpos^S(q,t)$ is undefined.

$$\frac{}{q = \bot^Q \lor t = \bot^S \lor \neg sor^S(q) \;\vdash\; sendpos^S(q,t) = \bot^I} \quad \text{(R 119)}$$

$$\frac{}{sendpos^S(q,t) = \bot^I \;\vdash\; q = \bot^Q \;,\; t = \bot^S \;,\; \neg sor^S(q)} \quad \text{(R 120)}$$

Receive Position for Random Access In the first rule, intuitively, F means that all elements in q up to and excluding the position i do not match the pattern $(t_1, \dots, t_r)$. G means that the element at position i matches the pattern. Those are precisely the conditions for $recpos^{\sigma,S}(q, t_1, \dots, t_r)$ to have the defined value i. With Length(S) $= n$ for $\sigma = \{i_1, \dots, i_r\} \subseteq [1, n]$ and $t_j \in Terms(S_{i_j})$ for $j \in [1; r]$:

$$\frac{\neg q = \bot^Q \;,\; \neg t_1 = \bot^{S_1} \;,\; \dots, \neg t_r = \bot^{S_{i_r}} \;,\; \neg i = \bot^I \;\vdash\; F \land G}{\neg q = \bot^Q \;,\; \neg t_1 = \bot^{S_1} \;,\; \dots \;,\; \neg t_r = \bot^{S_{i_r}} \;,\; \neg i = \bot^I \;\vdash\; recpos^{\sigma,S}(q, t_1, \dots, t_r) = i} \quad \text{(R 121)}$$

where

$$F = \forall x_1^I \big((c_1 \le x_1 < i) \to \neg match^{\sigma,S}(read^S(q, x_1), t_1, \dots, t_r)\big)$$

and

$$G = match^{\sigma,S}(read^S(q,i), t_1, \dots, t_r)$$

The following rules express the definedness condition. In particular H means that no element in q matches the pattern. In this case the value of $recpos^{\sigma,S}(q, t_1, \dots, t_r)$ is undefined.

$$\frac{}{q = \bot^Q \lor t_1 = \bot^{S_{i_1}} \lor \dots \lor t_r = \bot^{S_{i_r}} \lor H \;\vdash\; recpos^{\sigma,S}(q, t_1, \dots, t_r) = \bot^I} \quad \text{(R 122)}$$

$$\frac{}{recpos^{\sigma,S}(q, t_1, \dots, t_r) = \bot^I \;\vdash\; q = \bot^Q \;,\; t_1 = \bot^{S_{i_1}} \;,\; \dots \;,\; t_r = \bot^{S_{i_r}} \;,\; H} \quad \text{(R 123)}$$

where

$$H = \forall x_1^I \big(c_1 \le x_1 \le length(q) \to \neg match^{\sigma,S}(read^S(q, x_1), t_1, \dots, t_r)\big)$$

10.6.5 Predicate Symbols

10.6.5.1 Equality

General Properties For $S \in \Sigma$ and $t_1, t_2, t_3 \in Terms(S)$:

Reflexivity:

$$\frac{}{\vdash\ t_1 = t_1} \tag{R 124}$$

Symmetry:

$$\frac{}{t_1 = t_2\ \vdash\ t_2 = t_1} \tag{R 125}$$

Transitivity:

$$\frac{}{t_1 = t_2, t_2 = t_3\ \vdash\ t_1 = t_3} \tag{R 126}$$

The congruence axioms can be derived from the rule (R 21).

Composed Sorts For $S \in \Sigma_P$ with Length$(S) = n$ and $t_1, t_2 \in Terms(S)$:

$$\frac{\vdash\ elem_1^S(t_1) = elem_1^S(t_2) \wedge \ldots \wedge elem_n^S(t_1) = elem_n^S(t_2)}{\vdash\ t_1 = t_2} \tag{R 127}$$

$$\frac{elem_1^S(t_1) = elem_1^S(t_2) \wedge \ldots \wedge elem_n^S(t_1) = elem_n^S(t_2)\ \vdash}{t_1 = t_2\ \vdash} \tag{R 128}$$

Queues For $S \in \Sigma_P$ and $q_1, q_2 \in Terms(Q)$:

$$\frac{sor^S(q_1)\ \vdash\ cap(q_1) = cap(q_2) \wedge sor^S(q_2) \wedge length(q_1) = length(q_2) \wedge F}{sor^S(q_1)\ \vdash\ q_1 = q_2} \tag{R 129}$$

$$\frac{sor^S(q_1)\ ,\ cap(q_1) = cap(q_2) \wedge sor^S(q_2) \wedge length(q_1) = length(q_2) \wedge F\ \vdash}{sor^S(q_1)\ ,\ q_1 = q_2\ \vdash} \tag{R 130}$$

where

$$F = \forall x_1^I \big(c_1 \leq x_1 \leq length(q_1) \rightarrow read^S(q_1, x_1) = read^S(q_2, x_1)\big)$$

Processes

$$\frac{}{fin = run\ \vdash} \tag{R 131}$$

10.6.5.2 Comparison

General Properties For $S \in \Sigma_P$ and $t_1, t_2, t_3 \in Terms(S)$:

Reflexivity:

$$\frac{}{\vdash\ t_1 \leq t_1} \tag{R 132}$$

Anti-Symmetry:

$$\frac{}{t_1 \leq t_2, t_2 \leq t_1\ \vdash\ t_1 = t_2} \tag{R 133}$$

Transitivity:

$$\frac{}{t_1 \leq t_2, t_2 \leq t_3\ \vdash\ t_1 \leq t_3} \tag{R 134}$$

Totality (except for $\bot^S$):

$$\frac{}{\neg t_1 = \bot^S, \neg t_2 = \bot^S \ \vdash \ t_1 \leq t_2 \lor t_2 \leq t_1} \tag{R 135}$$

$\bot^S$ is only comparable to itself:

$$\frac{}{t_1 \leq \bot^S \lor \bot^S \leq t_1 \ \vdash \ t_1 = \bot^S} \tag{R 136}$$

Negated Comparison:

$$\frac{\vdash \ t_2 \leq t_1 \land \neg t_1 = t_2}{t_1 \leq t_2 \ \vdash} \tag{R 137}$$

Integers For $i, j, k \in \mathit{Terms}(I)$:

$$\frac{}{\vdash \ c_0 < c_1} \tag{R 138}$$

$$\frac{}{i < j \ , \ \neg k = \bot^I \ \vdash \ i + k < j + k} \tag{R 139}$$

$$\frac{}{i < j \ , \ c_0 < k \ \vdash \ i \cdot k < j \cdot k} \tag{R 140}$$

Channels For $c \in \mathit{Terms}(C)$:

$$\frac{}{\neg c = \bot^C \ \vdash \ c < \mathit{incchan}(c)} \tag{R 141}$$

Composed Sorts (Lexicographic Ordering) For $S \in \Sigma_P$ with Length$(S) = n$ and $t_1, t_2 \in \mathit{Terms}(S)$:

$$\frac{\vdash \ \bigvee_{i=1}^{n} \Big(\bigwedge_{j=1}^{i-1} \mathit{elem}_j^S(t_1) = \mathit{elem}_j^S(t_2) \land \mathit{elem}_i^S(t_1) \leq \mathit{elem}_i^S(t_2) \Big)}{\vdash \ t_1 \leq t_2} \tag{R 142}$$

10.6.5.3 Other Predicate Symbols

Pattern Matching If $\sigma = \{i_1, \ldots, i_r\}$, $S \in \Sigma_P$, $a \in \mathit{Terms}(S)$ and $b_j \in \mathit{Terms}(S_{i_j})$ for $j \in [1; r]$:

$$\frac{\vdash \ \neg a = \bot^S \land \bigwedge_{j=1}^{r} \mathit{elem}_{i_j}^S(a) = b_j}{\vdash \ \mathit{match}^{\sigma,S}(a, b_1, \ldots, b_r)} \tag{R 143}$$

$$\frac{\vdash \ a = \bot^S \ , \ \neg \mathit{elem}_{i_1}^S(a) = b_1 \ , \ \ldots \ , \ \neg \mathit{elem}_{i_r}^S(a) = b_r}{\mathit{match}^{\sigma,S}(a, b_1, \ldots, b_r) \ \vdash} \tag{R 144}$$

Sorts of Queues The idea of these rules is that the symbol sor^S can be pushed into its argument until an empty queue is met, the sort of which is known. While doing that, definedness conditions produce additional proof goals.

For $m \in \mathbb{N}$, $S, S' \in \Sigma_P$, $S \neq S'$:

$$\frac{}{\vdash \ \mathit{sor}^S(()^{m,S})} \tag{R 145}$$

$$\frac{}{\mathit{sor}^S(()^{m,S'}) \ \vdash} \tag{R 146}$$

$$\frac{}{\mathit{sor}^S(\bot^Q) \ \vdash} \tag{R 147}$$

For $S, S' \in \Sigma_P$, $q \in Terms(Q)$, $a \in Terms(S')$, $i \in Terms(I)$:

$$\frac{\vdash\ \neg a = \bot^S \wedge \neg i = \bot^I \wedge sor^S(q)}{\vdash\ sor^S(insert^{S'}(q, a, i))} \tag{R 148}$$

$$\frac{sor^S(q)\ \vdash\ a = \bot^S\ ,\ i = \bot^I}{sor^S(insert^{S'}(q, a, i))\ \vdash} \tag{R 149}$$

$$\frac{\vdash\ \neg i = \bot^I \wedge sor^S(q)}{\vdash\ sor^S(remove(q, i))} \tag{R 150}$$

$$\frac{sor^S(q)\ \vdash\ i = \bot^I}{sor^S(remove(q, i))\ \vdash} \tag{R 151}$$

10.6.6 Early Simplification

All rules for rigid function and predicate symbols contain only first-order formulas and no updates. This is sufficient for completeness because updates can be eliminated by rule (R 55). But it is more efficient to apply first-order simplification rules before the application of updates. That is called early simplification.

The early simplification rules arise from a normal rule by prefixing every formula in the antecedent or succedent of a premise or the conclusion with U and by adding the applicability condition that U is any finite sequence of updates. The original rule then becomes the special case where $U = ()$.

Clearly the global (strong) soundness of a rule implies the global (strong) soundness of the corresponding early simplification rule. However, in many cases the corresponding early simplification rule cannot be derived and has to be explicitly included in the calculus. This has not been done for simplicity. But early simplification is used in the examples and should be used in any implementation. In three important cases early simplification rules are part of **C**: the rules (R 21) (application of equality), (R 22) and (R 23) (conditional terms).

Another type of possible early simplification rules arises by using modal operators instead of sequences of updates. The most interesting example is the application of equality within the scope of a modal operator:

$$\frac{[[\pi]]t = t'\ \vdash\ M(\pi)F'}{[[\pi]]t = t'\ \vdash\ M(\pi)F}$$

where $M \in Oper$, $\pi \in Prog$ and t, t', F and F' are as in rule (R 21).

10.7 Examples

10.7.1 Continued Example

At this point the derivation of the example formulas should be given. However, even these very simple formulas are far too complex for the whole derivation to be printed. This shows both the power and the complexity of the automated theorem proving for DLTP. Therefore only the first formula, $F = [\text{RUN GLOBAL}]\ Y_1^I =^I c_{42}$ is derived under the precondition $P = \neg nextpid =^I \bot^I \wedge \neg nextchan =^C \bot^C$.

Several syntactical abbreviations are used:

- The abbreviations π_0 to π_{11} of section 8.3.1 are used again.
- The abbreviations in 8.3.1 for the updates cannot be used because those updates are interpreted. The syntactical versions of the updates U_0 to U_{10} as they appear in the calculus are as follows.

  $v_0 = (proc, nextpid, run) \cdot (var^I, (nextpid, c_1), c_{41}) \cdot (nextchan, (), nextchan) \cdot (nextpid, (), nextpid + c_1)$

  $v_1 = (proc, nextpid, run) \cdot (var^C, (nextpid, c_1), C_1) \cdot (var^C, (nextpid, c_2), C_2) \cdot (cont, C_1, ()^{1,C})\cdot$
  $(cont, C_2, ()^{1,I}) \cdot (nextchan, (), C_3) \cdot (nextpid, (), nextpid + c_1)$
  with $C_1 = nextchan$, $C_2 = incchan(nextchan)$ and $C_3 = incchan(incchan(nextchan))$

  $v_2 = (proc, nextpid, run) \cdot (var^C, (nextpid, c_1), ini^C) \cdot (nextchan, (), nextchan)\cdot$
  $\big(var^C, (nextpid, c_2), var^C(nextpid - c_1, c_1)\big) \cdot (nextpid, (), nextpid + c_1)$

$$v_3 = (proc, nextpid, run) \cdot (nextchan, (), nextchan) \cdot \big(var^C, (nextpid, c_1), var^C(nextpid - c_1 - c_1, c_2)\big) \cdot \\ (nextpid, (), nextpid + c_1)$$

$$v_4 = \Big(cont, X, insert^C\big(cont(X), var^C(nextpid - c_1 - c_1 - c_1, c_2), length(cont(X)) + c_1\big)\Big) \\ \text{with } X = var^C(nextpid - c_1 - c_1 - c_1, c_1)$$

$$v_5 = (proc, p, fin) \cdot (var^C, (p, c_1), \bot^C) \cdot (var^C, (p, c_2), \bot^C) \\ \text{with } p = nextpid - c_1 - c_1 - c_1$$

$$v_6 = \big(var^C, (nextpid - c_1 - c_1, c_1), elem_1^C(read^C(cont(X), c_1))\big) \cdot \big(cont, X, remove(cont(X), c_1)\big) \\ \text{with } X = var^C(nextpid - c_1 - c_1, c_2)$$

$$v_7 = (cont, X, insert^I(cont(X), c_{42}, c_1)) \\ \text{with } X = var^C(nextpid - c_1 - c_1, c_1)$$

$$v_8 = (proc, p, fin) \cdot (var^C, (p, c_1), \bot^C) \cdot (var^C, (p, c_2), \bot^C) \\ \text{with } p = nextpid - c_1 - c_1$$

$$v_9 = \big(var^I, (glob, c_1), elem_1^I(read^I(cont(X), c_1))\big) \cdot \big(cont, X, remove(cont(X), c_1)\big) \\ \text{with } X = var^C(nextpid - c_1, c_1)$$

$$v_{10} = (proc, p, fin) \cdot (var^C, (p, c_1), \bot^C) \\ \text{with } p = nextpid - c_1$$

These sequences of updates satisfy $v_i = \mathit{Eff}^A(c_i)$ where c_i is the command that is executed in the state g_i for $i \in [0; 10]$ where g_0 to g_{10} are as in section 8.3.1. Additionally let $V_i = v_0 \cdot \ldots \cdot v_i$ for $i \in [0; 10]$.

- For $i \in [0, 10]$, Ex_i abbreviates the formula that states that any first command of any process in π_i is asynchronously or synchronously executable. This formula is defined formally in rule (R 27), by the application of which it is introduced.
- For $i \in [0, 10]$ and $k \in \text{Length}(\pi_i)$, $Exec_i(k) = Exec^A(\text{First}(\pi_{ik}))$ states the asynchronous executability of the first command of the k-th process of the program π_i.
- The formula P is left out from the antecedents of all sequents except for the root, because it is not explicitly used in any of the shown steps of the proof. But it is needed in omitted steps.
- If a formula can be derived in a trivial way the proof goal is immediately closed. This is indicated by $\boxed{\text{C}}$ under the respective formula. This is used, for example, when the formula *true* occurs in the succedent, or when a scheduling decision has been made that requires the synchronous execution of two commands for which exec^S is defined as $false$.
- If there are several formulas in a sequent, only one of them must be derived. The other formulas (in the antecedent and the succedent) can be eliminated by the weakening rule (R 3). This is indicated by $\boxed{\text{W}}$ under the respective formula. This is used for formulas in the antecedent that specify the executability of a command that is executable.
- Proof goals that must be closed by deriving a formula that specifies that a command is not executable are left open. This is indicated with $\boxed{\text{O}}$ under the respective formula. These formulas do not contain modal operators, but their derivation still needs several mathematically simple, but notationally complex steps.

These abbreviations result in an almost linear proof tree: The root of the proof is at the top. And in every step there is only one sequent not marked with $\boxed{\text{C}}$ or $\boxed{\text{O}}$ and in this sequent there is only one formula not marked with $\boxed{\text{W}}$. This formula is underlined and the next rule is applied to this formula. Applying a rule means to find the conclusion of the rule and to replace it with the premises.[56] The applied rule is given indented between two sequents. The most often applied rules are (R 27) (scheduling), (R 41) (symbolic execution) and (R 55) (application of updates).

[56]This may be confusing: The proof is read top-to-bottom, but each rule is read bottom-to-top.

Root: $P \vdash \underline{[\pi_0]F}$

(R 27)

$\underset{\boxed{W}}{Ex_0} \vdash \underline{[1:\ \pi_0]F} \quad + \quad \neg Ex_0 \vdash \underset{\boxed{C}}{true}$

(R 41)

$\underset{\boxed{W}}{Exec_0(1)} \vdash \underline{V_0\ [\pi_1]F}$

(R 27)

$V_0\ \underset{\boxed{W}}{Ex_1} \vdash \underline{V_0\ [1:\ \pi_1]F} \quad + \quad V_0\ \neg Ex_1 \vdash \underset{\boxed{C}}{true}$

(R 41)

$V_0\ \underset{\boxed{W}}{Exec_1(1)} \vdash \underline{V_1\ [\pi_2]F}$

(R 27)

$V_1\ \underset{\boxed{W}}{Ex_2} \vdash \underline{V_1\ [1:\ \pi_2]F} \quad + \quad V_1\ \neg Ex_2 \vdash \underset{\boxed{C}}{true}$

(R 41)

$V_1\ \underset{\boxed{W}}{Exec_2(1)} \vdash \underline{V_2\ [\pi_3]F}$

(R 27)

$V_2\ \underset{\boxed{W}}{Ex_3} \vdash \underline{V_2\ [1:\ \pi_3]F} \quad + \quad V_2\ \underset{\boxed{W}}{Ex_3} \vdash V_2\ \underset{\boxed{O}}{[2:}\ \pi_3]F \quad + \quad V_2\ Ex_3 \vdash V_2\ [1,2:\ \underset{\boxed{C}}{\pi_3]F} \quad +$

$V_2\ \neg Ex_3 \vdash \underset{\boxed{C}}{true}$

(R 41)

$V_2\ \underset{\boxed{W}}{Exec_3(1)} \vdash \underline{V_3\ [\pi_4]F}$

(R 27)

$V_3\ \underset{\boxed{W}}{Ex_4} \vdash \underline{V_3\ [1:\ \pi_4]F} \quad + \quad V_3\ \underset{\boxed{W}}{Ex_4} \vdash V_3\ \underset{\boxed{O}}{[2:}\ \pi_4]F \quad + \quad V_3\ \underset{\boxed{W}}{Ex_4} \vdash V_3\ \underset{\boxed{O}}{[3:}\ \pi_4]F \quad +$

$V_3\ \underset{\boxed{W}}{Ex_4} \vdash V_2\ [1,\underset{\boxed{O}}{2:}\ \pi_4]F \quad + \quad V_3\ \underset{\boxed{W}}{Ex_4} \vdash V_2\ [1,\underset{\boxed{O}}{3:}\ \pi_4]F \quad + \quad V_3\ Ex_4 \vdash V_2\ [2,1:\ \underset{\boxed{C}}{\pi_4]F} \quad +$

$V_3\ Ex_4 \vdash V_2\ [2,\underset{\boxed{C}}{3:}\ \pi_4]F \quad + \quad V_3\ Ex_4 \vdash V_2\ [3,\underset{\boxed{C}}{1:}\ \pi_4]F \quad + \quad V_3\ Ex_4 \vdash V_2\ [3,2:\ \underset{\boxed{C}}{\pi_4]F} \quad +$

$V_3\ \neg Ex_4 \vdash \underset{\boxed{C}}{true}$

(R 41)

$\vdots$

$V_8\ \underset{\boxed{W}}{Exec_9(1)} \vdash \underline{V_9\ [\pi_{10}]F}$

(R 26) (termination of a process)

$\vdash \underline{V_9\ [1:\ \pi_{10}]F}$

(R 41)

$V_9\ \underset{\boxed{W}}{Exec_{10}(1)} \vdash \underline{V_{10}\ [\pi_{11}]F}$

(R 25) (termination of a program)

$\vdash \underline{V_{10}\ F}$

(R 55)

$$\vdash\ \sigma_{V_{10}}(F)$$
$$\boxed{\mathrm{O}}$$

To complete the example all formulas that are marked with $\boxed{\mathrm{O}}$ have to be derived. Inspecting the sequences V_0 to V_{10} of updates and the definition of update substitutions shows that such a proof would already be too complex to read it. Therefore only the simplest example, the formula $V_2\ [2 : \pi_3]F$, is derived. In the proof the following additional abbreviations are used:

- Several rule applications are omitted and referred to by "simplification". These steps eliminate conditional terms with trivial guards (e. g. $\mathit{nextpid} = \mathit{nextpid} \wedge c_2 = c_2$) and simplify arithmetic expressions (e. g. $\mathit{nextpid} + c_1 - c_1$). This may require a new branch in the proof which is immediately closed. In some cases the precondition P is needed, e. g. to derive $\neg \mathit{nextpid} =^I \mathit{nextpid} + c_1$.
- Early simplification is used.
- $c = \mathrm{First}(\pi_{3,2}) = \mathit{var}^C(\mathit{nextpid} - c_1, c_2)\ ?\ \emptyset\ (\mathit{var}^C(\mathit{nextpid} - c_1, c_1))\ ()$ is the first command of the second process of π_3.
- With $n = \mathrm{Length}(V_2)$ and for $i \in [1; n-1]$ let $V^i = \mathrm{Remove}(V_2, [i+1, n])$.

Then the proof is almost linear. Again the root of the proof is at the top and the applied rules are given in between.

Root: $P \vdash V_2\ [2 : \pi_3]F$

(R 41)

$V_2\ \mathit{Exec}^A(c) \vdash V_2 \cdot \mathit{Eff}^A(c)\ [\mathrm{rem}^{\mathrm{A}}(\pi_3, 2)]F$
$\boxed{\mathrm{W}}$

Definition of $\mathit{Exec}^A(c)$

$V_2\ \mathit{pollfirst}^{\emptyset,C}(\mathit{var}^C(\mathit{nextpid} - c_1, c_2)) =^I c_1 \vdash$

(R 64) (definable non-rigid function symbol)

$V_2\ \big[\mathit{match}^{\emptyset,C}\big(\mathit{read}^C(\mathit{cont}(\mathit{var}^C(\mathit{nextpid} - c_1, c_2)), c_1)\big)\ ?\ c_1\ :\ c_0\big] =^I c_1 \vdash$

Simplification[57]

$V_2\ \mathit{match}^{\emptyset,C}\Big(\mathit{read}^C\big(\mathit{cont}(\mathit{var}^C(\mathit{nextpid} - c_1, c_2)), c_1\big)\Big) \vdash$

(R 144) (pattern matching)

$\vdash V_2\ \mathit{read}^C\big(\mathit{cont}(\mathit{var}^C(\mathit{nextpid} - c_1, c_2)), c_1\big) =^C \bot^C$

(R 55) for $u = (\mathit{nextpid}, (), \mathit{nextpid} + c_1)$

$\vdash V^{15}\ \mathit{read}^C\big(\mathit{cont}(\mathit{var}^C(\mathit{nextpid} + c_1 - c_1, c_2)), c_1\big) =^C \bot^C$

Simplification

$\vdash V^{15}\ \mathit{read}^C\big(\mathit{cont}(\mathit{var}^C(\mathit{nextpid}, c_2)), c_1\big) =^C \bot^C$

(R 55) for $u = (\mathit{var}^C, (\mathit{nextpid}, c_2), \mathit{var}^C(\mathit{nextpid} - c_1, c_1))$ and simplification

$\vdash V^{14}\ \mathit{read}^C\big(\mathit{cont}(\mathit{var}^C(\mathit{nextpid} - c_1, c_1)), c_1\big) =^C \bot^C$

(R 55) for $u = (\mathit{nextchan}, (), \mathit{nextchan})$, $u = (\mathit{var}^C, (\mathit{nextpid}, c_1), \mathit{ini}^C)$, $u = (\mathit{proc}, \mathit{nextpid}, \mathit{run})$, $u = (\mathit{nextpid}, (), \mathit{nextpid} + c_1)$ and $u = (\mathit{nextchan}, (), \mathit{incchan}(\mathit{incchan}(\mathit{nextchan})))$ and simplification

$\vdash V^9\ \mathit{read}^C\big(\mathit{cont}(\mathit{var}^C(\mathit{nextpid}, c_1)), c_1\big) =^C \bot^C$

(R 55) for $u = (\mathit{cont}, \mathit{incchan}(\mathit{nextchan}), ()^{1,I})$

$\vdash V^8\ \mathit{read}^C\Big(\big[\mathit{var}^C(\mathit{nextpid}, c_1) = \mathit{incchan}(\mathit{nextchan})\ ?\ ()^{1,I}\ :\ \mathit{cont}(\mathit{var}^C(\mathit{nextpid}, c_1))\big], c_1\Big) =^C \bot^C$

(R 55) for $u = (\mathit{cont}, \mathit{nextchan}, ()^{1,C})$

$$\begin{aligned}\vdash V^7\ \mathit{read}^C\Big(&\big[\mathit{var}^C(\mathit{nextpid}, c_1) = \mathit{incchan}(\mathit{nextchan})\ ?\\ &()^{1,I}\ :\ [\mathit{var}^C(\mathit{nextpid}, c_1) = \mathit{nextchan}\ ?\ ()^{1,C}\ :\ \mathit{cont}(\mathit{var}^C(\mathit{nextpid}, c_1))]\big]\\ &, c_1\Big) =^C \bot^C\end{aligned}$$

(R 55) for $u = (\mathit{var}^C, (\mathit{nextpid}, c_2), \mathit{incchan}(\mathit{nextchan}))$ and $u = (\mathit{var}^C, (\mathit{nextpid}, c_1), \mathit{nextchan})$ and simplification

$$\begin{aligned}\vdash V^5\ \mathit{read}^C\Big(&\big[\mathit{nextchan} = \mathit{incchan}(\mathit{nextchan})\ ?\\ &()^{1,I}\ :\ [\mathit{nextchan} = \mathit{nextchan}\ ?\ ()^{1,C} : \mathit{cont}(\mathit{nextchan})]\big]\\ &, c_1\Big) =^C \bot^C\end{aligned}$$

Simplification (eliminating the outer conditional term)

$\vdash V^5\ \mathit{read}^C\big([\mathit{nextchan} = \mathit{nextchan}\ ?\ ()^{1,C}\ :\ \mathit{cont}(\mathit{nextchan})], c_1\big) =^C \bot^C$

Simplification (eliminating the conditional term)

$\vdash V^5\ \mathit{read}^C(()^{1,C}, c_1) =^C \bot^C$

(R 55) for the remaining updates, rule (R 2) with the appropriate instance of the formula in the antecedent of the rule (R 116); then the first proof goal is closed by rule (R 116), the second by several rules in particular (R 98).

[57]Here rule (R 2) (cut) is applied with the formula $V_2\ \mathit{match}^{\emptyset,C}(\mathit{read}^C(\mathit{cont}(\mathit{var}^C(\mathit{nextpid} - c_1, c_2)), c_1))$, which results in two branches. The rest of the given proof shows the first branch. The second branch is closed by applying the rules (R 23) (conditional term), (R 21) (application of equality) and integer axioms to disprove the equality of 0 and 1.

10.7.2 An Induction Proof

Let $X = Y_1^I = var^I(glob, c_1)$, $x = x_1^I$ and

```
c = DO
      :: X; X := X − c_1
      :: ELSE; BREAK
   OD
```

π does not terminate from all states and termination may take arbitrarily many steps. Therefore π cannot be eliminated directly.

The following is a proof of $c_0 \leq^I X \vdash [c]X =^I c_0$ that uses the induction hypothesis

$$\forall x(c_0 \leq^I x =^I X \to [c]X =^I c_0).$$

Again the root is at the top. In some cases the part to which a rule is applied is underlined, and the used rules are given in between. The most often used rules are:

- (R 1): Self-proof
- (R 3): Weakening
- (R 4) and (R 5): Truth
- (R 27): Scheduling
- (R 31): Unwinding
- (R 41): Symbolic execution
- The following derived rules for formulas F, G, H:

  $$\mathbf{A}: \frac{\vdash F + \vdash G + H \vdash}{F \wedge G \to H \vdash}$$

  and

  $$\mathbf{B}: \frac{F, G \vdash H}{\vdash F \wedge G \to H}$$

The proof is not linear: Branches are indicated by itemizations.

Root: $c_0 \leq^I X \ \vdash\ [c]X =^I c_0$

(R 2) (cut) with $F = \forall x(c_0 \leq^I x =^I X \to [c]X =^I c_0)$

$\boxed{1} \quad + \quad \boxed{2}$

– Proof goal $\boxed{1}$: $\underline{\forall x(c_0 \leq^I x =^I X \to [c]X =^I c_0)}\ ,\ c_0 \leq^I X \ \vdash\ [c]X =^I c_0$

(R 18) (universal quantifier) with $t = X$, (R 3), rule A

$\underline{c_0 \leq^I X} \ \vdash\ \underline{c_0 \leq^I X}\ ,\ [c]X =^I c_0 \quad +$
$c_0 \leq^I X \ \vdash\ \underline{X =^I X}\ ,\ [c]X =^I c_0 \quad +$
$c_0 \leq^I X\ ,\ \underline{[c]X =^I c_0} \ \vdash\ \underline{[c]X =^I c_0}$

closed by rules (R 1), (R 124) and (R 1), respectively

– Proof goal $\boxed{2}$: $c_0 \leq^I X \ \vdash\ \underline{\forall x(c_0 \leq^I x =^I X \to [c]X =^I c_0)}\ ,\ [c]X =^I c_0$

twice (R 3), (R 14) (universal quantifier), (R 88) (induction)

$\boxed{2.1} \quad + \quad \boxed{2.2} \quad + \quad \boxed{2.3} \quad + \quad \boxed{2.4}$

- Proof goal $\boxed{2.1}$: $\vdash\ \underline{(c_0 \leq^I c_0 =^I X \to [c]X =^I c_0)}$

  rule B, (R 3)

  $c_0 =^I X \ \vdash\ [\underline{c}]X =^I c_0$

  (R 27) with $Ex = true$

  $c_0 =^I X\ ,\ \underline{true} \ \vdash\ [\underline{1:\ c}]X =^I c_0 \quad + \quad c_0 =^I X\ ,\ \neg true \ \vdash\ \underline{true}$

  first proof goal: (R 5), (R 31) with $Else = \neg\neg X =^I ini^I$; second proof goal: closed by (R 4)

  $c_0 =^I X \ \vdash\ [1:\ \underline{X};\ X := X - c_1;\ \mathrm{BT}\ c]X =^I c_0 \quad +$
  $c_0 =^I X\ ,\ \neg\neg X =^I ini^I \ \vdash\ [1:\ \underline{\mathrm{ELSE}};\ \mathrm{BREAK};\ \mathrm{BT}\ c]X =^I c_0$

  second proof goal: (R 3); both proof goals: (R 41)

  $c_0 =^I X\ ,\ \neg X =^I ini^I \ \vdash\ [X := X - c_1;\ \mathrm{BT}\ c]X =^I c_0$
  $c_0 =^I X\ ,\ \underline{true} \ \vdash\ [\underline{\mathrm{BREAK};\ \mathrm{BT}\ c}]X =^I c_0$

  first proof goal: closed by (R 69) (initial value for sort I) and integer axioms;
  second proof goal: (R 5), (R 27) with $Ex = true$

  $c_0 =^I X\ ,\ \underline{true} \ \vdash\ [1:\ \underline{\mathrm{BREAK}};\ \mathrm{BT}\ c]X =^I c_0 \quad + \quad c_0 =^I X\ ,\ \neg true \ \vdash\ \underline{true}$

  first proof goal: (R 5), (R 41), (R 5); second proof goal: closed by (R 4)

  $c_0 =^I X \ \vdash\ [\underline{\epsilon}]X =^I c_0$

  (R 25) (termination of programs)

  $c_0 =^I X \ \vdash\ X =^I c_0$

  closed by (R 125)

- Proof goal $\boxed{2.2}$: $c_0 \leq^I x\ ,\ c_0 \leq^I x =^I X \to [c]X =^I c_0 \ \vdash\ \underline{c_0 \leq^I (x + c_1) =^I X \to [c]X =^I c_0}$

  rule B

  $c_0 \leq^I x\ ,\ \underline{c_0 \leq^I x =^I X \to [c]X =^I c_0}\ ,\ c_0 \leq^I x + c_1\ ,\ x + c_1 =^I X \ \vdash\ [c]X =^I c_0$

  rule A

$\underline{c_0 \leq^I x}\ ,\ c_0 \leq^I x + c_1\ ,\ x + c_1 =^I X\ \vdash\ \underline{c_0 \leq^I x}\ ,\ [c]X =^I c_0 \quad +$
$c_0 \leq^I x\ ,\ \underline{c_0 \leq^I x + c_1}\ ,\ x + c_1 =^I X\ \vdash\ x =^I X\ ,\ [c]X =^I c_0 \quad +$
$c_0 \leq^I x\ ,\ c_0 \leq^I x + c_1\ ,\ x + c_1 =^I X\ ,\ \underline{[c]X =^I c_0}\ \vdash\ \underline{[c]X =^I c_0}$

first and third proof goal: closed by (R 1); second proof goal: (R 3)

$c_0 \leq^I x\ ,\ x + c_1 =^I X\ \vdash\ x =^I X\ ,\ [\underline{c}]X =^I c_0$

(R 27) with $Ex = true$

$c_0 \leq^I x\ ,\ x + c_1 =^I X\ ,\ \underline{true}\ \vdash\ x =^I X\ ,\ [\underline{1:\ c}]X =^I c_0 \quad +$
$c_0 \leq^I x\ ,\ x + c_1 =^I X\ ,\ \neg true\ \vdash\ x =^I X\ ,\ \underline{true}$

first proof goal: (R 5), (R 31) with $Else = \neg\neg X =^I ini^I$; second proof goal: closed by (R 4)

$\boxed{2.2.1} \quad + \quad \boxed{2.2.2}$

∗ Proof goal $\boxed{2.2.1}$: $c_0 \leq^I x\ ,\ x + c_1 =^I X\ \vdash\ x =^I X\ ,\ [1:\ \underline{X};\ X := X - c_1;\ \text{BT}\ c]X =^I c_0$

(R 3), (R 41)

$x + c_1 =^I X\ ,\ \neg X =^I ini^I\ \vdash\ x =^I X\ ,\ [\underline{X := X - c_1;\ \text{BT}\ c}]X =^I c_0$

(R 3), (R 27) with $Ex = true$

$x + c_1 =^I X\ ,\ \underline{true}\ \vdash\ x =^I X\ ,\ [1:\ \underline{X := X - c_1};\ \text{BT}\ c]X =^I c_0 \quad +$
$x + c_1 =^I X\ ,\ \neg true\ \vdash\ x =^I X\ ,\ \underline{true}$

first proof goal: (R 5), (R 41), (R 5); second proof goal: closed by (R 4)

$\underline{x + c_1 =^I X}\ \vdash\ x =^I X\ ,\ (var^I, (glob, c_1), \underline{X - c_1})[c]X =^I c_0$

(R 21), (R 3)

$\vdash\ x =^I X\ ,\ \underline{(var^I, (glob, c_1), x + c_1 - c_1)}[c]X =^I c_0$

arithmetic simplification and (R 56)

$\vdash\ x =^I X\ ,\ \underline{X =^I x \to [c]X =^I c_0}$

rule B (without G)

$\underline{X =^I x}\ \vdash\ \underline{x =^I X}\ ,\ [c]X =^I c_0$

closed by (R 125)

∗ Proof goal $\boxed{2.2.2}$: $c_0 \leq^I x\ ,\ x{+}c_1 =^I X\ ,\ \underline{\neg\neg X =^I ini^I}\ \vdash\ x =^I X\ ,\ [1:\ \text{ELSE};\ \text{BREAK};\ \text{BT}\ c]X =^I c_0$

twice (R 3), (R 9), (R 8)

$c_0 \leq^I x\ ,\ x + c_1 =^I X\ ,\ X =^I ini^I\ \vdash$

closed by (R 69) and integer axioms

- Proof goal $\boxed{2.3}$: $x \leq^I c_0\ ,\ c_0 \leq^I x =^I X \to [c]X =^I c_0\ \vdash\ \underline{c_0 \leq^I (x - c_1) =^I X \to [c]X =^I c_0}$

rule B, three times (R 3)

$x \leq^I c_0\ ,\ c_0 \leq^I x - c_1\ \vdash$

closed by integer axioms

- Proof goal $\boxed{2.4}$: $\vdash\ \underline{c_0 \leq^I \bot^I =^I X \to [c]X =^I c_0}$

rule B, twice (R 3)

$c_0 \leq^I \bot^I \ \vdash$

closed by (R 136) and (R 68)

11 Soundness

11.1 Result

In this section soundness means soundness for the standard structure if no structure is mentioned.

Theorem 1. *The following soundness results hold:*

- *The rule* (R 3) *(weakening) is locally sound for all states $g \in \mathbf{G}$.*
- *The rules* (R 48) *and* (R 57) *(generalization for modal operators and updates)*[58] *and* (R 56) *(indirect elimination of updates) are globally sound.*
- *All other rules of* **C** *are locally strongly sound for all states $g \in \mathbf{G}$.*

Proof. The proof is given in this section, separately for each rule. The local strong soundness for all states of the rule (R 45) is rather a completeness result and is therefore proven in Theorem 2 in the next section. □

Using this theorem and the following criterion, the weaker, but more important result follows that **C** is globally sound.

Soundness Criterion The proofs of this section will implicitly use the following soundness criterion which shows that the context of a rule can be ignored in the proof of local strong soundness, provides a formally simple definition of local strong soundness and shows that local soundness implies global soundness.

Lemma 4. *Let K be a structure (for any vocabulary) with set of states G, $G' \subseteq G$, let R be a rule with premises $\{\Gamma, A_i \vdash S_i, \Delta | i \in [1;n]\}$ and conclusion $\Gamma, A \vdash S, \Delta$ for sets of formulas A_i, S_i for $i \in [1;n]$ and A, S, Γ, Δ. Then:*

- *If*

$$\inf_{i\in[1;n]} \overline{\mathrm{val}_g^\alpha}(A_i \vdash S_i) = \overline{\mathrm{val}_g^\alpha}(A \vdash S)$$

holds for a state $g \in G$ and all assignments α, then R is locally strongly sound for K and g.

- *If R is locally (strongly) sound for K and all $g \in G'$, then R is (strongly) sound for K and G'.*

Proof.

- For the bottom-up direction assume $\overline{\mathrm{val}_g}(\Gamma, A \vdash S, \Delta) = 1$. Let α be an arbitrary assignment, two cases are distinguished:
  - $\overline{\mathrm{val}_g^\alpha}(\Gamma \vdash \Delta) = 1$: Then immediately $\overline{\mathrm{val}_g^\alpha}(\Gamma, A_i \vdash S_i, \Delta) = 1$ for $i \in [1;n]$.
  - $\overline{\mathrm{val}_g^\alpha}(\Gamma \vdash \Delta) = 0$: Then Γ and Δ drop out from all sequents and by the assumption $\overline{\mathrm{val}_g^\alpha}(A \vdash S) = 1$. Then the prerequisites imply that $\overline{\mathrm{val}_g^\alpha}(A_i \vdash S_i) = 1$ for $i \in [1;n]$.

  Therefore $\overline{\mathrm{val}_g}(\Gamma, A_i \vdash S_i, \Delta) = 1$ for $i \in [1;n]$.

  The proof of the top-down direction is similar.
- The result follows immediately from the definitions of local and global strong soundness.

□

[58] See the proof for stronger results that hold if F is rigid.

11.2 Structural Rules and First-Order Predicate Logic

Classical Rules The local strong soundness for all states of the structural rules and the rules for first-order logic ((R 1) to (R 20)) except for rule (R 3) and the local soundness of rule (R 3) are standard results (see for example [11]).

Application of Equality The local strong soundness of rule (R 21) for all states can be reduced to the case $U = ()$. Then it is a standard result except for one case: Let $F = u\ G$ for an update $u = (f, (r_1 \dots, r_n), r)$, then the applicability conditions allow to replace an occurrence of t in $r_1, \dots, r_n$ or r with t'; let u' be an update that arises from u by doing this. This case, where F may be in the antecedent or in the succedent, is treated in the following.

Let $g \in \mathbf{G}$ and an assignment α be arbitrary, and let $g' = \mathit{State}(g, \alpha, U)$. If $\overline{\mathrm{val}_g^\alpha}(U\ t = t') = 0$, then clearly the semantics of the premise and the conclusion is equal. Therefore the result reduces to $\overline{\mathrm{val}_g^\alpha}(U\ u'\ G) = \overline{\mathrm{val}_g^\alpha}(U\ u\ G)$ under the condition that $\overline{\mathrm{val}_g^\alpha}(U\ t = t') = 1$.

To show this let $g_1'' = \mathit{State}(g', \alpha, u)$ and $g_2'' = \mathit{State}(g', \alpha, u')$. Then

$$\overline{\mathrm{val}_g^\alpha}(U\ u'\ G) \overset{\boxed{1}}{=} \overline{\mathrm{val}_{g'}^\alpha}(u'\ G) \overset{\boxed{2}}{=} \overline{\mathrm{val}_{g_2''}^\alpha}(G) \overset{\boxed{3}}{=} \overline{\mathrm{val}_{g_1''}^\alpha}(G) \overset{\boxed{4}}{=} \overline{\mathrm{val}_{g'}^\alpha}(u\ G) \overset{\boxed{5}}{=} \overline{\mathrm{val}_g^\alpha}(U\ u\ G)$$

$\boxed{1}$ by the definition of g'

$\boxed{2}$ by the definition of g_2''

$\boxed{3}$ because $\overline{\mathrm{val}_g^\alpha}(U\ t = t') = \overline{\mathrm{val}_{g'}^\alpha}(t = t') = 1$ implies that $g_1'' = g_2''$

$\boxed{4}$ by the definition of g_1''

$\boxed{5}$ by the definition of g'

Conditional Terms Local strong soundness for all $g \in \mathbf{G}$ of the rules (R 22) and (R 23) can be reduced to the case $U = ()$ without loss of generality.

For that case let $g \in \mathbf{G}$ and α be an assignment. Two cases are distinguished:

- $\overline{\mathrm{val}_g^\alpha}(C) = 1$: Then $\overline{\mathrm{val}_g^\alpha}(T) = \overline{\mathrm{val}_g^\alpha}(t)$ and therefore
  - (R 22): $\overline{\mathrm{val}_g^\alpha}(\neg C) = 0 = \overline{\mathrm{val}_g^\alpha}(\neg T =^S t \lor \neg C)$
  - (R 23): $\overline{\mathrm{val}_g^\alpha}(C) = 1 = \overline{\mathrm{val}_g^\alpha}(\neg T =^S t' \lor C)$
- $\overline{\mathrm{val}_g^\alpha}(C) = 0$: Then $\overline{\mathrm{val}_g^\alpha}(T) = \overline{\mathrm{val}_g^\alpha}(t')$ and therefore
  - (R 22): $\overline{\mathrm{val}_g^\alpha}(\neg C) = 1 = \overline{\mathrm{val}_g^\alpha}(\neg T =^S t \lor \neg C)$
  - (R 23): $\overline{\mathrm{val}_g^\alpha}(C) = 0 = \overline{\mathrm{val}_g^\alpha}(\neg T =^S t' \lor C)$

From that local strong soundness of both rules for all $g \in \mathbf{G}$ follows.

11.3 Elimination of Programs

11.3.1 Preliminary Results

Asynchronous Executability

Lemma 5. *For all states g of the standard structure and all commands c:*

$$\overline{\mathrm{val}_g}(\mathit{Exec}^A(c)) = 1 \text{ if and only if } g \in \mathrm{exec}^A(c).$$

Proof. By structural induction over c. For all $g \in \mathbf{G}$: If

- $c \in \underline{\text{assignment}} \cup \underline{\text{break}} \cup \underline{\text{else}} \cup \underline{\text{run}} \cup \underline{\text{end}}$:

  $$\overline{\mathrm{val}}_g(Exec^A(c)) = \overline{\mathrm{val}}_g(true) = 1$$

  if and only if

  $$g \in \mathbf{G} = \mathrm{exec}^{\mathrm{A}}(c)$$

- $c \in \underline{\text{expression}}$, $c = t$ for $t \in Terms(S)$:

  $$\overline{\mathrm{val}}_g(Exec^A(c)) = \overline{\mathrm{val}}_g(\neg t =^S ini^S) = 1$$

  if and only if

  $$\overline{\mathrm{val}}_g(t) \neq \overline{\mathrm{val}}_g(ini^S)$$

  if and only if $g \in \mathrm{exec}^{\mathrm{A}}(c)$

- $c \in \underline{\text{send}}$, $Sor(c) = S$, $Chan(C) = C$: If $\overline{\mathrm{val}}_g\big(cont(C)\big) = \bot$, then $\overline{\mathrm{val}}_g(Exec^A(c)) = false$ and $\mathrm{exec}^{\mathrm{A}}(c) = \emptyset$.

  If $\overline{\mathrm{val}}_g\big(cont(C)\big) = (m, S', q)$:

  $$\overline{\mathrm{val}}_g(Exec^A(c)) = \overline{\mathrm{val}}_g\big(lengthc(C) <^I cap(cont(C)) \wedge sor^S(cont(C))\big) = 1$$

  if and only if

  $$length(q) < m \text{ and } S = S'$$

  if and only if $g \in \mathrm{exec}^{\mathrm{A}}(c)$.

- $c \in \underline{\text{receive}}$, $Sor(c) = S$, $Chan(C) = C$, $Pos(c) = \sigma = \{i_1, \ldots, i_r\}$, $Arg(c) = (a_i)_{i\in\sigma}$ and

  - $Type(c) \in \{?, ?<\}$:

    $$\overline{\mathrm{val}}_g(Exec^A(c)) = \overline{\mathrm{val}}_g\big(pollfirst^{\sigma,S}(C, a_{i_1}, \ldots, a_{i_r}) =^I c_1\big) = 1$$

    if and only if

    $$\overline{\mathrm{val}}_g\big(recpos^{\sigma,S}(cont(C), a_{i_1}, \ldots, a_{i_r})\big) = 1$$

    if and only if $g \in \mathrm{exec}^{\mathrm{A}}(c)$

  - $Type(c) \in \{??, ??<\}$:

    $$\overline{\mathrm{val}}_g(Exec^A(c)) = \overline{\mathrm{val}}_g\big(pollany^{\sigma,S}(C, a_{i_1}, \ldots, a_{i_r}) =^I c_1\big) = 1$$

    if and only if

    $$\overline{\mathrm{val}}_g\big(recpos^{\sigma,S}(cont(C), a_{i_1}, \ldots, a_{i_r})\big) \neq \bot$$

    if and only if $g \in \mathrm{exec}^{\mathrm{A}}(c)$

- $c \in \underline{\text{if}} \cup \underline{\text{do}}$ and

  - $optelse(c) = 0$:

    $$\overline{\mathrm{val}}_g(Exec^A(c)) = \overline{\mathrm{val}}_g\Big(\bigvee_{i=1}^{opt(c)} Exec^A(c[i]_1)\Big) = 1$$

    if and only if

    $$\overline{\mathrm{val}}_g\big(Exec^A(c[i]_1)\big) = 1 \text{ for some } i \in [1; opt(c)]$$

    if and only if, using the induction hypothesis for $c[i]_1$ for $i \in [1; opt(c)]$,

    $$g \in \bigcup_{i=1}^{opt(c)} \mathrm{exec}^{\mathrm{A}}(c[i]_1) = \mathrm{exec}^{\mathrm{A}}(c)$$

  - $optelse(c) = 1$:

    $$\overline{\mathrm{val}_g}\big(Exec^A(c)\big) = \overline{\mathrm{val}_g}(true) = 1$$

    if and only if, using the induction base for ELSE,

    $$g \in \mathbf{G} = \mathrm{exec}^{\mathrm{A}}(c)$$

- $c \in \underline{\text{atomic}}$:

  $$\overline{\mathrm{val}_g}(Exec^A(c)) = \overline{\mathrm{val}_g}\big(Exec^A(c[body]_1)\big) = 1$$

  if and only if, using the induction hypothesis for $c[body]_1$,

  $$g \in \mathrm{exec}^{\mathrm{A}}(c[body]_1) = \mathrm{exec}^{\mathrm{A}}(c)$$

□

Synchronous Executability

Lemma 6. *For all states g of the standard structure and all commands c and d:*

$\overline{\mathrm{val}_g}(Exec^S(c,d)) = 1$ if and only if $g \in \mathrm{exec}^{\mathrm{S}}(c,d)$.

Proof. By structural induction over c and d. For all $g \in \mathbf{G}$: If

- $c \in \underline{\text{send}}$, $d \in \underline{\text{receive}}$, $Chan(c) = C$, $Chan(d) = C'$, $Sor(c) = S$, $Sor(d) = S'$, $Pos(d) = \sigma = \{i_1, \ldots, i_r\}$ and $Arg(d) = (a_i)_{i\in\sigma}$:

  $$\overline{\mathrm{val}_g}(Exec^S(c,d)) = \overline{\mathrm{val}_g}(E) = 1$$

  where

  $$E = \; C =^{C} C' \;\wedge\; cap(cont(C)) =^{I} c_0 \;\wedge\; sor^S(cont(C))$$
  $$\wedge\; recpos^{\sigma,S'}\big(insert^S\big(()^{1,S}, Arg(c), c_1\big), a_{i_1}, \ldots, a_{i_r}\big) =^{I} c_1$$

  and

  $$\mathrm{exec}^{\mathrm{S}}(c,d) = G$$

  where

  $$G = \Big\{g \in \mathbf{G} | \mathrm{val}_g(C) = \mathrm{val}_g(C'),\; \overline{\mathrm{val}_g}(cont(C)) = (0, S, ()),\; S = S',$$
  $$\overline{\mathrm{val}_g}\Big(match^{\sigma,S}\big(Arg(c), a_{i_1}, \ldots, a_{i_r}\big)\Big) = 1\Big\}$$

  The equivalence follows from

  - $\overline{\mathrm{val}_g}(C =^{C} C') = 1$ if and only if $\mathrm{val}_g(C) = \mathrm{val}_g(C')$
  - $\overline{\mathrm{val}_g}\big(cap(cont(C)) =^{I} c_0\big) = 1$ and $\overline{\mathrm{val}_g}\big(sor^S(cont(C))\big) = 1$ if and only if $\overline{\mathrm{val}_g}(cont(C)) = (0, S, ())$
  - $\overline{\mathrm{val}_g}\Big(recpos^{\sigma,S'}\big(insert^S\big(()^{1,S}, Arg(c), c_1\big), a_{i_1}, \ldots, a_{i_r}\big) =^{I} c_1\Big) = 1$ if and only if $S = S'$ and $\overline{\mathrm{val}_g}\Big(match^{\sigma,S}\big(Arg(c), a_{i_1}, \ldots, a_{i_r}\big)\Big) = 1$

- otherwise and $c, d \in \underline{\text{elementary}}$: The result follows from $\overline{\mathrm{val}_g}(SynExec(c,d)) = 0$ and $\mathrm{exec}^{\mathrm{S}}(c,d) = \emptyset$.
- $c \notin \underline{\text{elementary}}$ or $d \notin \underline{\text{elementary}}$: The proof proceeds essentially as in the asynchronous case (where it is used that commands of the type $\underline{\text{else}}$ are never synchronously executable).

□

Effect

Lemma 7. *For all states $g \in \mathbf{G}$ and*

- *all elementary commands c such that $g \in \mathrm{exec}^{\mathrm{A}}(c)$:*

$$\overline{\mathrm{val}_g}(\mathit{Eff}^A(c)) * \mathrm{val}_g = \mathrm{val}_{\mathrm{eff}^{\mathrm{A}}{}_c(g)}$$

- *all elementary commands c, d such that $g \in \mathrm{exec}^{\mathrm{S}}(c, d)$:*

$$\overline{\mathrm{val}_g}(\mathit{Eff}^S(c, d)) * \mathrm{val}_g = \mathrm{val}_{\mathrm{eff}^{\mathrm{S}}{}_{c,d}(g)}$$

Proof. $\mathrm{val}_{\mathrm{eff}^{\mathrm{A}}{}_c(g)}$ and $\mathrm{val}_{\mathrm{eff}^{\mathrm{S}}{}_{c,d}(g)}$ are defined by applying a sequence of interpreted updates to val_g. The results follow if the interpretation of the updates in $\mathit{Eff}^A(c)$ and $\mathit{Eff}^S(c, d)$ gives those interpreted updates. That can be directly verified. □

Case Distinctions for Modal Operators The following lattice operations that depend on the modal operators capture the necessary case distinctions in several soundness proofs.

For $M \in \mathit{Oper}$:

$$(\diamondsuit_M, \diamond_M) = \begin{cases} (\inf, \wedge) & \text{if } M \in \mathit{Oper}^U \\ (\sup, \vee) & \text{if } M \in \mathit{Oper}^E \end{cases}$$

For $M \in \mathit{Oper}$ and $a, b \in \mathbf{B}$:

$$\mathrm{lat}_M(a, b) = \begin{cases} b & \text{if } M \in \mathit{Oper}_F \\ a \wedge b & \text{if } M \in \mathit{Oper}_U \\ a \vee b & \text{if } M \in \mathit{Oper}_E \end{cases}$$

The remaining lemmas state the properties of these operations that are needed in the soundness proofs. These properties hold for all structures.

Unions of Sets of Traces This lemma captures the fact that the modal operators $M \in \mathit{Oper}^U$ quantify universally and the modal operators $M \in \mathit{Oper}^E$ quantify existentially over traces.

Lemma 8. *For all $M \in \mathit{Oper}$, all families of sets of traces $(T_i)_{i \in I}$, all assignments α and all formulas F:*

$$\mathrm{SemModal}(M, \bigcup_{i \in I} T_i, \alpha, F) = \diamondsuit_{M\, i \in I} \mathrm{SemModal}(M, T_i, \alpha, F)$$

Proof. The proof is clear after putting in the definition of SemModal because for $B_i \subseteq \mathbf{B}$ for $i \in I$ and $\diamondsuit \in \{\inf, \sup\}$:

$$\diamondsuit \bigcup_{i \in I} B_i = \diamondsuit\{\diamondsuit B_i | i \in I\}.$$

□

Propagation of the Modal Operator along the Traces The next lemma states that the value of $M(i : \pi)F$ in the state g can be reduced to the value of F in g and the value of $M(\pi')F$ in g', where g' is the state and π' the remaining program after execution of $\mathrm{First}(\pi_i)$.

The main idea is that the traces in $\overline{\mathrm{val}_g}(i : \pi)$ differ from those in $\overline{\mathrm{val}_{g'}}(\pi')$ only by having the additional first element g. Therefore the quantification over the intermediate states of a trace only differs by including the state g which can be done by the lattice operation lat_M.

Lemma 9. *If*

1. *α is an assignment,*
2. *$\pi \in \mathit{Prog}$,*

3. $i \in [1; \mathrm{Length}(\pi)]$,

4. $c = \mathrm{First}(\pi_i) \in \underline{\mathrm{elementary}}$,

5. $\gamma \in \mathrm{exec}^{\mathrm{A}}(c)$,

and with the abbreviations

- $R = \big\{\mathrm{First}(\pi_j) | j \in [1; \mathrm{Length}(\pi)] \setminus \{i\}\big\}$,
- $\gamma' = \mathrm{eff}^{\mathrm{A}}{}_c(\gamma)$,
- $\pi' = \mathrm{rem}^{\mathrm{A}}(\pi, i)$,

then

$$\overline{\mathrm{val}_\gamma^\alpha}\big(M(i:\ \pi)F\big) = \mathrm{lat}_M\Big(\overline{\mathrm{val}_\gamma^\alpha}(F)\ ,\ \overline{\mathrm{val}_{\gamma'}^\alpha}(M(\pi')F)\Big).$$

Proof. Let $V = \overline{\mathrm{val}_\gamma}(i:\ \pi)$ and $V' = \overline{\mathrm{val}_{\gamma'}}(\pi')$. Using the assumptions 4 and 5 yields $\mathrm{Unwind}^{\mathrm{A}}(\pi_i, \gamma, R) = \{\pi_i\}$ and then the definition of $\overline{\mathrm{val}_\gamma}(i:\ \pi)$ reduces to $V = \{(\gamma) \cdot h | h \in V'\}$. Let

$$S = \overline{\mathrm{val}_\gamma^\alpha}\big(M(i:\ \pi)F\big) = \mathrm{SemModal}(M, V, \alpha, F).$$

Putting in the definition of SemModal yields

- if $M \in Oper_F$:

$$S = \underset{t\in V, \eta \in Fin(t)}{\Diamond_M} \overline{\mathrm{val}_\eta^\alpha}(F) \overset{\boxed{1}}{=} \underset{t\in V', \eta \in Fin(t)}{\Diamond_M} \overline{\mathrm{val}_\eta^\alpha}(F) \overset{\boxed{2}}{=} \overline{\mathrm{val}_{\gamma'}^\alpha}(M(\pi')F)$$

- if $M \in Oper_U$:

$$S = \underset{t\in V}{\Diamond_M} \inf_{\eta \in IntMed(t)} \overline{\mathrm{val}_\eta^\alpha}(F) \overset{\boxed{3}}{=} \underset{t\in V}{\Diamond_M}\big(\overline{\mathrm{val}_\gamma^\alpha}(F) \wedge \inf_{\eta\in IntMed(\mathrm{Rest}(t))} \overline{\mathrm{val}_\eta^\alpha}(F)\big)$$

$$\overset{\boxed{4}}{=} \overline{\mathrm{val}_\gamma^\alpha}(F) \wedge \underset{t\in V}{\Diamond_M} \inf_{\eta\in IntMed(\mathrm{Rest}(t))} \overline{\mathrm{val}_\eta^\alpha}(F) \overset{\boxed{5}}{=} \overline{\mathrm{val}_\gamma^\alpha}(F) \wedge \underset{t\in V'}{\Diamond_M} \inf_{\eta\in IntMed(t)} \overline{\mathrm{val}_\eta^\alpha}(F)$$

$$\overset{\boxed{6}}{=} \overline{\mathrm{val}_\gamma^\alpha}(F) \wedge \overline{\mathrm{val}_{\gamma'}^\alpha}(M(\pi')F)$$

- if $M \in Oper_E$:

$$S \overset{\boxed{7}}{=} \overline{\mathrm{val}_\gamma^\alpha}(F) \vee \overline{\mathrm{val}_{\gamma'}^\alpha}(M(\pi')F)$$

Then all three cases can be combined to

$$S = \mathrm{lat}_M\big(\overline{\mathrm{val}_\gamma^\alpha}(F), \overline{\mathrm{val}_{\gamma'}^\alpha}(M(\pi')F)\big).$$

□

$\boxed{1}$ because $\bigcup_{t\in V} Fin(t) = \bigcup_{t\in V'} Fin(t)$

$\boxed{2}$ by the semantics of $M(\pi')F$

$\boxed{3}$ by the associative law

$\boxed{4}$ by the distributive (if $\Diamond_M = \sup$) or associative (if $\Diamond_M = \inf$) law

$\boxed{5}$ because $V = \{(\gamma) \cdot h | h \in V'\}$

$\boxed{6}$ by the semantics of $M(\pi')F$

$\boxed{7}$ by the same arguments as in the preceding case

The following lemma states the same as Lemma 9, but for the case of synchronous execution.

Lemma 10. *If*

1. α *is an assignment*
2. $\pi \in Prog$,
3. $i, j \in [1; \mathrm{Length}(\pi)]$, $i \neq j$,
4. $c = \mathrm{First}(\pi_i) \in \underline{\mathrm{elementary}}$,
5. $d = \mathrm{First}(\pi_j) \in \underline{\mathrm{elementary}}$,
6. $\gamma \in \mathrm{exec}^{\mathrm{S}}(c, d)$,

and with the abbreviations

- $\gamma' = \mathrm{eff^S}_{c,d}(\gamma)$,
- $\pi' = \mathrm{rem}^{\mathrm{S}}(\pi, i, j)$,

then

$$\overline{\mathrm{val}}_\gamma^\alpha\big(M(i,j:\ \pi)F\big) = \mathrm{lat}_M\Big(\overline{\mathrm{val}}_\gamma^\alpha(F)\ ,\ \overline{\mathrm{val}}_{\gamma'}^\alpha(M(\pi')F)\Big).$$

Proof. The proof is essentially the same as for Lemma 9. □

11.3.2 Duality

The rule (R 24) for the duality of modal operators is locally strongly sound for all states because of the duality of universal and existential quantification over a Boolean lattice. This duality holds because for $A \subseteq \mathbf{B}$ and $A' = \{\neg a | a \in A\}$: $\neg \inf\ A = \sup A'$ and $\neg \sup A = \inf A'$, which implies

$$\overline{\mathrm{val}}_g^\alpha(M(\pi)F) = \overline{\mathrm{val}}_g^\alpha(\neg M(\pi)\neg F)$$

for all interpretation functions val_g and all assignments α.

11.3.3 Scheduling

Let $g \in \mathbf{G}$ and an assignment α be arbitrary, and let

- $g' = State(g, \alpha, U)$,
- $T_i = \overline{\mathrm{val}}_{g'}(\pi)$,
- $A_i = \overline{\mathrm{val}}_{g'}(i:\ \pi)$ for $i \in [1; n]$,
- $S_{ij} = \overline{\mathrm{val}}_{g'}(i, j:\ \pi)$ for $i, j \in [1; n]$, $i \neq j$.

Termination of Programs Local strong soundness for g of the rule (R 25) follows from $\overline{\mathrm{val}}_g^\alpha(U\ M(\epsilon)F) = \overline{\mathrm{val}}_g^\alpha(U\ F)$ and that holds because:

$$\overline{\mathrm{val}}_g^\alpha(U\ M(\epsilon)F) \overset{\boxed{1}}{=} \overline{\mathrm{val}}_{g'}^\alpha(M(\epsilon)F) \overset{\boxed{2}}{=} \mathrm{SemModal}(M, \{(\underline{\mathrm{termination}}\ g')\}, \alpha, F) \overset{\boxed{3}}{=} \overline{\mathrm{val}}_{g'}^\alpha(F) \overset{\boxed{4}}{=} \overline{\mathrm{val}}_g^\alpha(U\ F)$$

$\boxed{1}$ by the definition of g'
$\boxed{2}$ by the definition of $\overline{\mathrm{val}}_{g'}(\epsilon)$
$\boxed{3}$ by the definition of SemModal
$\boxed{4}$ by the definition of g'

Termination of Processes Local strong soundness for g of the rule (R 26) follows from $\overline{\mathrm{val}}_g^\alpha(U\ M(\pi)F) = \overline{\mathrm{val}}_g^\alpha(U\ M(i:\ \pi)F)$ and that holds because:

$$\overline{\mathrm{val}}_g^\alpha(U\ M(\pi)F) \overset{\boxed{1}}{=} \mathrm{SemModal}(M, \overline{\mathrm{val}_{g'}}(\pi), \alpha, F) \overset{\boxed{2}}{=} \mathrm{SemModal}(M, T_i, \alpha, F)$$

$$\overset{\boxed{3}}{=} \mathrm{SemModal}(M, A_i, \alpha, F) \overset{\boxed{4}}{=} \mathrm{SemModal}(M, \overline{\mathrm{val}_{g'}}(i:\ \pi), \alpha, F) \overset{\boxed{5}}{=} \overline{\mathrm{val}}_g^\alpha(U\ M(i:\ \pi)F)$$

$\boxed{1}$ by the definition of g' and SemModal

$\boxed{2}$ by the definition of $\overline{\mathrm{val}_{g'}}(\pi)$ because $\mathrm{First}(\pi_k) \notin \underline{\mathrm{end}}$ for $k \in [1; i-1]$ and $\mathrm{First}(\pi_i) \in \underline{\mathrm{end}}$

$\boxed{3}$ $T_i = A_i$ because commands of the type $\underline{\mathrm{end}}$ are never synchronously executable[59]

$\boxed{4}$ by the definition of A_i

$\boxed{5}$ by the definition of g' and SemModal

No Exclusive Control and Timeouts To prove local strong soundness for g of the rules (R 27) and (R 28) two cases are distinguished. By the Lemmas 5 and 6 $\overline{\mathrm{val}_{g'}}(Ex) = 1$ holds precisely if $\bigcup_{i=1}^{n} T_i \neq \emptyset$ holds. Then if

- $\overline{\mathrm{val}_{g'}}(Ex) = 1$: The sequent with $U\ \neg Ex$ in the antecedent drops out and $U\ Ex$ drops out from all antecedents. Then

  $$\overline{\mathrm{val}}_g^\alpha(U\ M(\pi)F) \overset{\boxed{1}}{=} \mathrm{SemModal}(M, \overline{\mathrm{val}_{g'}}(\pi), \alpha, F)$$

  $$\overset{\boxed{2}}{=} \mathrm{SemModal}(M, \bigcup_{i=1}^{n} T_i, \alpha, F) \overset{\boxed{3}}{=} \mathop{\diamondsuit_M}_{i\in[1;n]} \mathrm{SemModal}(M, T_i, \alpha, F)$$

  Using

  $$\mathrm{SemModal}(M, T_i, \alpha, F) \overset{\boxed{4}}{=} \mathrm{SemModal}(M, A_i, \alpha, F) \diamond_M \mathop{\diamondsuit_M}_{j\in[1;n]\setminus\{i\}} \mathrm{SemModal}(M, S_{ij}, \alpha, F)$$

  for $i \in [1; n]$, yields

  $$\overline{\mathrm{val}}_g^\alpha(U\ M(\pi)F) = \mathop{\diamondsuit_M}_{i\in[1;n]} \overline{\mathrm{val}}_g^\alpha(U\ M(i:\ \pi)F) \diamond_M \mathop{\diamondsuit_M}_{\substack{i,j\in[1;n]\\ i\neq j}} \overline{\mathrm{val}}_g^\alpha(U\ M(i,j:\ \pi)F)$$

  and for both rules putting in the definition of $\diamondsuit_M$ and $\diamond_M$ yields what local strong soundness for g requires. $\boxed{5}$

- $\overline{\mathrm{val}_{g'}}(Ex) = 0$: The sequents with $U\ Ex$ in the antecedent drop out and $U\ \neg Ex$ drops out from all antecedents. Then for both rules local strong soundness for g requires $\overline{\mathrm{val}}_g^\alpha(U\ M(\pi)F) = \overline{\mathrm{val}}_g^\alpha(T)$. This holds because $\overline{\mathrm{val}_g}(U\ M(\pi)F) \overset{\boxed{6}}{=} \mathrm{SemModal}(M, \{(\underline{\mathrm{timeout}}\ g')\}, \alpha, F) \overset{\boxed{7}}{=} \left\{\begin{array}{ll} 1 & \text{if } M = [] \\ 0 & \text{if } M = \langle\rangle \\ \overline{\mathrm{val}}_{g'}^\alpha(F) & \text{otherwise} \end{array}\right\} \overset{\boxed{8}}{=} \overline{\mathrm{val}}_g^\alpha(T)$.

$\boxed{1}$ by the definition of g' and SemModal

$\boxed{2}$ by the definition of $\overline{\mathrm{val}_{g'}}(\pi)$ because $\mathrm{First}(\pi_i) \notin \underline{\mathrm{end}}$, $\mathrm{Label}(\mathrm{First}(\pi_i)) \neq \mathrm{EC}$ for $i \in [1; n]$ and $\bigcup_{i=1}^{n} T_i \neq \emptyset$

$\boxed{3}$ by Lemma 8

$\boxed{4}$ by Lemma 8 and the definition of T_i

$\boxed{5}$ because several formulas in the succedent correspond to taking the supremum (if $M \in \mathit{Oper}^E$), and several premises to taking the infimum of the truth values (if $M \in \mathit{Oper}^U$)

$\boxed{6}$ by the definition of $\overline{\mathrm{val}_{g'}}(\pi)$ because $\pi \neq \epsilon$ and $\bigcup_{i=1}^{n} T_i = \emptyset$

$\boxed{7}$ because $\mathit{Fin}((\underline{\mathrm{timeout}}\ g')) = \emptyset$ and $\mathit{IntMed}((\underline{\mathrm{timeout}}\ g')) = \{g'\}$

$\boxed{8}$ by the definition of T and g'

---

[59]That implies that this rule may have to be modified for other structures.

Exclusive Control To prove local strong soundness for g of the rules (R 29) and (R 30) two cases are distinguished. By the Lemmas 5 and 6 $\overline{\text{val}_{g'}}(Ex) = 1$ holds precisely if $T_I \neq \emptyset$ holds. Then if:

- $\overline{\text{val}_{g'}}(Ex) = 1$: The sequent with $U\ \neg Ex$ in the antecedent drops out and $U\ Ex$ drops out from all antecedents. Then the proof proceeds as in the case without exclusive control with the difference that $\overline{\text{val}_{g'}}(\pi) = T_I$ because $\text{Label}(\text{First}(\pi_I)) = \text{EC}$ and $T_I \neq \emptyset$.
- $\overline{\text{val}_{g'}}(Ex) = 0$: The sequents with $U\ Ex$ in the antecedent drop out and $U\ \neg Ex$ drops out from all antecedents. Then for both rules local strong soundness for g requires

$$\overline{\text{val}_g^\alpha}(U\ M(\pi)F) = \overline{\text{val}_g^\alpha}(U\ M(\text{Dropec}(\pi))F).$$

  And that holds due to the definition of $\overline{\text{val}_{g'}}(\pi)$ in the case where $\text{Label}(\text{First}(\pi_I)) = \text{EC}$ and $T_I = \emptyset$.

11.3.4 Unwinding

11.3.4.1 Unwinding with Respect to Asynchronous Execution

Let $g \in \mathbf{G}$ and an assignment α be arbitrary, and let $g' = \mathit{State}(g, \alpha, U)$ and $R = \big\{\text{First}(\pi_j) | j \in [1;n] \setminus \{i\}\big\}$.

Selections To prove local strong soundness for g of the rules (R 31) and (R 32) two cases are distinguished. By the Lemmas 5 and 6

$$\overline{\text{val}_{g'}}(\mathit{Else}) = 1$$

holds precisely if

$$\text{Unwind}^{\text{A}}(c[j], g', R) = \emptyset \text{ and } \text{Unwind}^{\text{S}}(c[j], \rho, g') = \emptyset \text{ for } j \in [1;r] \text{ and } \rho \in R.$$

Then if:

1. $\mathit{optelse}(c) = 1$ and $\overline{\text{val}_{g'}}(\mathit{Else}) = 1$:
   Then the sequent $U\ \mathit{Else}\ \vdash\ U\ N$ reduces to $\vdash\ U\ M(i:\ \pi^i(u_{r+1}))F$ and the formula $U\ (\mathit{Else} \wedge N)$ reduces to $U\ M(i:\ \pi^i(u_{r+1}))F$.
   Then

$$\overline{\text{val}_{g'}}(i:\ \pi) \overset{\boxed{1}}{=} \big\{(g')\cdot h | t \in \text{Unwind}^{\text{A}}(s, g', R),\ g'' = \text{eff}^{\text{A}}{}_{t_1}(g'),\ h \in \overline{\text{val}_{g''}}(\text{rem}^{\text{A}}(\pi^i(t), i))\big\}$$

$$\overset{\boxed{2}}{=} \big\{(g')\cdot h | t \in \bigcup_{j=1}^{r+1} \text{Unwind}^{\text{A}}(u_j, g', R),\ g'' = \text{eff}^{\text{A}}{}_{t_1}(g'),\ h \in \overline{\text{val}_{g''}}(\text{rem}^{\text{A}}(\pi^i(t), i))\big\}$$

$$\overset{\boxed{3}}{=} \bigcup_{j=1}^{r+1} \big\{(g')\cdot h | t \in \text{Unwind}^{\text{A}}(u_j, g', R),\ g'' = \text{eff}^{\text{A}}{}_{t_1}(g'),\ h \in \overline{\text{val}_{g''}}(\text{rem}^{\text{A}}(\pi^i(t), i))\big\}$$

$$\overset{\boxed{4}}{=} \bigcup_{j=1}^{r+1} \overline{\text{val}_{g'}}(i : \pi^i(u_j))$$

   and therefore

$$\overline{\text{val}_g^\alpha}(U\ M(i:\ \pi)F) \overset{\boxed{5}}{=} \overline{\text{val}_{g'}^\alpha}(M(i:\ \pi)F) \overset{\boxed{6}}{=} \text{SemModal}(M, \overline{\text{val}_{g'}}(i:\ \pi), \alpha, F)$$

$$\overset{\boxed{7}}{=} \Diamond_{M\ j\in[1;r+1]} \text{SemModal}\big(M, \overline{\text{val}_{g'}}(i : \pi^i(u_j)), \alpha, F\big) \overset{\boxed{8}}{=} \Diamond_{M\ j\in[1;r+1]} \overline{\text{val}_{g'}^\alpha}(M(i : \pi^i(u_j))F)$$

$$\overset{\boxed{9}}{=} \Diamond_{M\ j\in[1;r+1]} \overline{\text{val}_g^\alpha}(U\ M(i : \pi^i(u_j))F)$$

   which is what local strong soundness for g requires for both rules.

2. $optelse(c) = 0$ or $\overline{\mathrm{val}_{g'}}(Else) = 0$:
   In both cases (in the first case by the definition of N) the sequent $U\ Else \vdash U\ N$ (if $M \in Oper^U$) or the formula $U\ (Else \wedge N)$ (if $M \in Oper^E$) drops out from the rules and also $else(c, g', R) = \emptyset$.
   Then
   $$\overline{\mathrm{val}_{g'}}(M(i:\ \pi)) = \bigcup_{j=1}^{r} \overline{\mathrm{val}_{g'}}(i : \pi^i(u_j))$$
   and
   $$\overline{\mathrm{val}_{g}^{\alpha}}(U\ M(i:\ \pi)F) = \underset{i\in[1;r]}{\Diamond_M} \overline{\mathrm{val}_{g}^{\alpha}}(U\ M(i : \pi^i(u_j))F),$$
   which is what local strong soundness for g requires for both rules, follow in essentially the same way as in the preceding case.

Both cases together yield local strong soundness for g both for $M \in Oper^U$ and for $M \in Oper^E$.

$\boxed{1}$ by the semantics of $i : \pi$
$\boxed{2}$ by the definition of $\mathrm{Unwind}^{\mathrm{A}}$ with $else(c, g', R) = \mathrm{Unwind}^{\mathrm{A}}(c[r+1], g', R)$
$\boxed{3}$ by set theory
$\boxed{4}$ by the semantics of $i : \pi^i(u_j)$ for $j \in [1; r+1]$
$\boxed{5}$ by the definition of g'
$\boxed{6}$ by the definition of SemModal
$\boxed{7}$ by the above and Lemma 8
$\boxed{8}$ by the definition of SemModal
$\boxed{9}$ by the definition of g'

Atomic Sequences

- $c \in \underline{\mathrm{if}} \cup \underline{\mathrm{do}}$: Let L be the following mapping between sets of command sequences
  $$L(M) = \bigcup_{t\in M} \mathrm{Unwind}^{\mathrm{A}}\big(\mathrm{ATOMIC}\{t\}; \mathrm{Rest}(S)\ ,\ g'\ ,\ R\big)$$
  The idea of the rules (R 33) and (R 34) is that first the selections in the body of the atomic sequence are unwound and when the case $c \in \underline{\mathrm{elementary}} \cup \underline{\mathrm{atomic}}$ is reached the first command is pulled out by using the rule (R 35). L captures the details of this last unwinding step, which are irrelevant for the soundness of the first steps.
  Apart from three additional steps that deal with L the proof of local strong soundness for g of the rules (R 33) and (R 34) proceeds in the same way as for the rules (R 31) and (R 32). The core of the proof is, if for example $optelse(c) = 1$ and $\overline{\mathrm{val}_{g'}}(Else) = 1$:
  $$\overline{\mathrm{val}_{g'}}(i:\ \pi) = \big\{(g')\cdot h | t \in \mathrm{Unwind}^{\mathrm{A}}(S, g', R),\ g'' = \mathrm{eff^A}_{t_1}(g'),\ h \in \overline{\mathrm{val}_{g''}}(\mathrm{rem}^{\mathrm{A}}(\pi^i(t), i))\big\}$$
  $$\overset{\boxed{1}}{=} \big\{(g')\cdot h | t \in L(\mathrm{Unwind}^{\mathrm{A}}(s, g', R)),\ g'' = \mathrm{eff^A}_{t_1}(g'),\ h \in \overline{\mathrm{val}_{g''}}(\mathrm{rem}^{\mathrm{A}}(\pi^i(t), i))\big\}$$
  $$= \big\{(g')\cdot h | t \in L\big(\bigcup_{j=1}^{r+1} \mathrm{Unwind}^{\mathrm{A}}(u_j, g', R)\big),\ g'' = \mathrm{eff^A}_{t_1}(g'),\ h \in \overline{\mathrm{val}_{g''}}(\mathrm{rem}^{\mathrm{A}}(\pi^i(t), i))\big\}$$
  $$\overset{\boxed{2}}{=} \big\{(g')\cdot h | t \in \bigcup_{j=1}^{r+1} L(\mathrm{Unwind}^{\mathrm{A}}(u_j, g', R)),\ g'' = \mathrm{eff^A}_{t_1}(g'),\ h \in \overline{\mathrm{val}_{g''}}(\mathrm{rem}^{\mathrm{A}}(\pi^i(t), i))\big\}$$
  $$\overset{\boxed{3}}{=} \big\{(g')\cdot h | t \in \bigcup_{j=1}^{r+1} \mathrm{Unwind}^{\mathrm{A}}(U_j, g', R),\ g'' = \mathrm{eff^A}_{t_1}(g'),\ h \in \overline{\mathrm{val}_{g''}}(\mathrm{rem}^{\mathrm{A}}(\pi^i(t), i))\big\}$$
  $$= \bigcup_{j=1}^{r+1} \big\{(g')\cdot h | t \in \mathrm{Unwind}^{\mathrm{A}}(U_j, g', R),\ g'' = \mathrm{eff^A}_{t_1}(g'),\ h \in \overline{\mathrm{val}_{g''}}(\mathrm{rem}^{\mathrm{A}}(\pi^i(t), i))\big\}$$
  $$= \bigcup_{j=1}^{r+1} \overline{\mathrm{val}_{g'}}(i : \pi^i(U_j))$$

- $c \in \underline{\text{elementary}} \cup \underline{\text{atomic}}$: To prove local strong soundness for g of the rule (R 35) first
$$\text{Unwind}^{\text{A}}(S, g', R) = \text{Unwind}^{\text{A}}(S', g', R)$$
holds by the definition of Unwind^{A}. That yields
$$\overline{\text{val}_{g'}}(i :\ \pi) = \overline{\text{val}_{g'}}(i :\ \pi^i(S'))$$
and therefore
$$\overline{\text{val}_g^\alpha}(U\ M(i :\ \pi)F) = \overline{\text{val}_g^\alpha}\big(U\ M(i :\ \pi^i(S'))F\big).$$

$\boxed{1}$ $\text{Unwind}^{\text{A}}(S, g', R) = L(\text{Unwind}^{\text{A}}(s, g', R))$ by the definitions of L and Unwind^{A} because $S = \text{ATOMIC}\{s\}; \text{Rest}(S)$

$\boxed{2}$ L is distributive over unions of sets

$\boxed{3}$ $\text{Unwind}^{\text{A}}(U_j, g', R) = L(\text{Unwind}^{\text{A}}(u_j, g', R))$ by the definitions of L and Unwind^{A} because $U_j = \text{ATOMIC}\{u_j\}; \text{Rest}(S)$ for $j \in [1; r + 1]$

11.3.4.2 Unwinding with Respect to Synchronous Execution

Selections The proof proceeds essentially as in the asynchronous case. It is much simpler because the else options can be ignored.[60] The core of the proof is, if for example $k = i$,

$$\overline{\text{val}_{g'}}(i, j :\ \pi) = \Big\{(g') \cdot h | (t, t') \in \text{Unwind}^{\text{S}}(s, \pi_j, g'),\ g'' = \text{eff}^{\text{S}}{}_{t_1, t_1'}(g'),\ h \in \overline{\text{val}_{g''}}\big(\text{rem}^{\text{S}}\big((\pi^i(t))^j(t'), i, j\big)\big)\Big\}$$
$$= \bigcup_{l=1}^{r} \Big\{(g') \cdot h | (t, t') \in \text{Unwind}^{\text{S}}(u_l, \pi_j, g'),\ g'' = \text{eff}^{\text{S}}{}_{t_1, t_1'}(g'),\ h \in \overline{\text{val}_{g''}}\big(\text{rem}^{\text{S}}\big((\pi^i(t))^j(t'), i, j\big)\big)\Big\}$$
$$= \bigcup_{l=1}^{r} \overline{\text{val}_{g'}}(i, j :\ \pi^i(u_l))$$

Atomic Sequences Again the proof proceeds essentially as in the asynchronous case.

11.3.5 Symbolic Execution

For the proof the rules are used in the form given in the table at the end of section 10.3.5. In particular the same abbreviations are used and the asynchronous and the synchronous case are treated together. If there are differences the formulation for the synchronous case is given in square brackets.

Let $g \in \mathbf{G}$ and an assignment α be arbitrary, and let $g' = \mathit{State}(g, \alpha, U)$.

By Lemma 5 [6] $\overline{\text{val}_g^\alpha}(E) = 1$ holds precisely if $g' \in \text{exec}^{\text{A}}(c)$ $[g' \in \text{exec}^{\text{S}}(c, d)]$. Two cases are distinguished:

- $\overline{\text{val}_g^\alpha}(E) = 1$: All sequents that have $\neg E$ in the antecedent drop out and E drops out of every antecedent. Then
  - for $M \in \mathit{Oper}_F$ the rules reduce to $\dfrac{\Gamma\ \vdash\ C', \Delta}{\Gamma\ \vdash\ C, \Delta}$,
  - for $M \in \mathit{Oper}_U$ the rules reduce to $\dfrac{\Gamma\ \vdash\ C', \Delta + \Gamma\ \vdash\ A, \Delta}{\Gamma\ \vdash\ C, \Delta}$,
  - for $M \in \mathit{Oper}_E$ the rules reduce to $\dfrac{\Gamma\ \vdash\ C', A, \Delta}{\Gamma\ \vdash\ C, \Delta}$.

  And local strong soundness for g requires $\overline{\text{val}_g^\alpha}(C) = \text{lat}_M(\overline{\text{val}_g^\alpha}(A), \overline{\text{val}_g^\alpha}(C'))$.

  Also $g' \in \text{exec}^{\text{A}}(c)$ $[g' \in \text{exec}^{\text{S}}(c, d)]$. Then Lemma 7 yields $\text{eff}^{\text{A}}{}_c(g') = \mathit{State}(g, \alpha, U \cdot \mathit{Eff}^{\,A}(c))$ $[\text{eff}^{\text{S}}{}_{c,d}(g') = \mathit{State}(g, \alpha, U \cdot \mathit{Eff}^{\,S}(c, d))]$. That and the applicability conditions allow to apply Lemma 9 [10] (with $\gamma = g'$), which yields the required condition.

[60] Therefore these rules have to be modified if more general structures are used.

- $\overline{\mathrm{val}}_g^\alpha(E) = 0$: All sequents that have E in the antecedent drop out and $\neg E$ drops out of every antecedent. Also $g' \notin \mathrm{exec}^{\mathrm{A}}(c)$ $[g' \notin \mathrm{exec}^{\mathrm{S}}(c,d)]$, which implies $\mathrm{Unwind}^{\mathrm{A}}(\pi_i, g', R) = \emptyset$ (for any $R \subseteq$ command) $[\mathrm{Unwind}^{\mathrm{S}}(\pi_i, \pi_j, g') = \emptyset]$. That yields $\overline{\mathrm{val}}_{g'}(i : \pi) = \emptyset$ $[\overline{\mathrm{val}}_{g'}(i, j : \pi) = \emptyset]$.

  Then:

  - For $M \in \mathit{Oper}^U$ the rules reduce to

    $$\frac{}{\Gamma \vdash C, \Delta}$$

    and local strong soundness for g requires $\overline{\mathrm{val}}_g^\alpha(\Gamma \vdash C, \Delta) = 1$. That holds because M quantifies universally over the empty set of traces, which implies $\overline{\mathrm{val}}_g^\alpha(C) = 1$.

  - For $M \in \mathit{Oper}^E$ the rules reduce to

    $$\frac{\Gamma \vdash \Delta}{\Gamma \vdash C, \Delta}$$

    and local strong soundness for g requires $\overline{\mathrm{val}}_g^\alpha(\Gamma \vdash C, \Delta) = \overline{\mathrm{val}}_g^\alpha(\Gamma \vdash \Delta)$. That holds because M quantifies existentially over the empty set of traces, which implies $\overline{\mathrm{val}}_g^\alpha(C) = 0$.

Both cases together yield local strong soundness for g.

11.4 Additional Rules for Modal Operators

The proofs of local strong soundness for all $g \in \mathbf{G}$ of the rules (R 46) and (R 47) are simple.

The global soundness of the rule (R 48) holds because for all assignments α, $\overline{\mathrm{val}}_g^\alpha(F) = 1$ for all $g \in \mathbf{G}$ implies that F holds for all states over which M quantifies. The restriction $M \neq \langle\rangle$ is necessary because it is not clear whether a final state exists.

If F is rigid then $\overline{\mathrm{val}}_g^\alpha(F)$ does not depend on the state and rule (R 48) is locally sound. If additionally $M \neq []$ rule (R 48) is locally strongly sound.

11.5 Updates

Updates and First-Order Logic The proofs of local strong soundness for all $g \in \mathbf{G}$ of the rules (R 49) to (R 54) are simple and similar. As an example the proof for rule (R 53) in the case $Q = \forall$ is shown.

Let $g \in \mathbf{G}$ and an assignment α be arbitrary, and let $g' = \mathit{State}(g, \alpha, u)$ and $x \in \mathit{LV}(S)$ for $S \in \Sigma$, then local strong soundness for g follows from:[61]

$$\overline{\mathrm{val}}_g^\alpha(u\ \forall x\ F) \overset{\boxed{1}}{=} \overline{\mathrm{val}}_{g'}^\alpha(\forall x\ F) \overset{\boxed{2}}{=} \inf_{v \in U(S)} \overline{\mathrm{val}}_{g'}^{\alpha_x^v}(F) \overset{\boxed{3}}{=} \inf_{v \in U(S)} \overline{\mathrm{val}}_{\mathit{State}(g, \alpha_x^v, u)}^{\alpha_x^v}(F)$$

$$\overset{\boxed{4}}{=} \inf_{v \in U(S)} \overline{\mathrm{val}}_g^{\alpha_x^v}(u\ F) \overset{\boxed{5}}{=} \overline{\mathrm{val}}_g^\alpha(\forall x\ u\ F)$$

Application of Updates Local strong soundness for all states of rule (R 55) follows immediately from the update substitution lemma.

Indirect Elimination of Updates The following argument shows global soundness of rule (R 56). Let I abbreviate $f(t_1, \ldots, t_n) =^S t$.

Assume $\overline{\mathrm{val}}_g(\Gamma \ \vdash \ u\ F\ ,\ \Delta) = 1$ for all $g \in \mathbf{G}$. Let $g \in \mathbf{G}$ and an assignment α be arbitrary and let $g' = \mathit{State}(g, \alpha, u)$. Then two cases are distinguished:

- $\overline{\mathrm{val}}_g^\alpha(\Gamma \vdash \Delta) = 1$: Then immediately $\overline{\mathrm{val}}_g^\alpha(\Gamma \vdash I \to F\ ,\ \Delta) = 1$.

[61] The update substitution lemma cannot be used in the proofs because F and G do not have to be first-order.

- $\overline{\mathrm{val}}_g^\alpha(\Gamma \vdash \Delta) = 0$: Then $\overline{\mathrm{val}}_g^\alpha(\Gamma \vdash I \to F \;,\; \Delta) = \overline{\mathrm{val}}_g^\alpha(I \to F)$ and the assumption implies $\overline{\mathrm{val}}_g^\alpha(u\ F) = 1$. Two further cases are distinguished:
  - $\overline{\mathrm{val}}_g^\alpha(I) = 0$: Then immediately $\overline{\mathrm{val}}_g^\alpha(I \to F) = 1$.
  - $\overline{\mathrm{val}}_g^\alpha(I) = 1$: Then $\overline{\mathrm{val}}_g^\alpha(F) \overset{\boxed{6}}{=} \overline{\mathrm{val}}_{g'}^\alpha(F) \overset{\boxed{7}}{=} 1$ and therefore $\overline{\mathrm{val}}_g^\alpha(I \to F) = 1$.

Therefore $\overline{\mathrm{val}}_g^\alpha(\Gamma \vdash I \to F \;,\; \Delta) = 1$ for all $g \in \mathbf{G}$ and all assignments α.

Update Generalization The rule (R 57) is globally sound:
If $\overline{\mathrm{val}}_g^\alpha(F) = 1$ for all $g \in \mathbf{G}$ and all assignments α, then for all assignments α $\overline{\mathrm{val}}_g^\alpha(u\ F) = \overline{\mathrm{val}}_{State(g,\alpha,u)}^\alpha(F) = 1$ for all $g \in \mathbf{G}$.

If F is rigid, then $\overline{\mathrm{val}}_g^\alpha(u\ F) = \overline{\mathrm{val}}_g^\alpha(F)$ for all $g \in \mathbf{G}$ and all assignments α, and the rule is locally strongly sound.

$\boxed{1}$ by the definition of g'
$\boxed{2}$ by the semantics of $\forall x\ F$
$\boxed{3}$ by the definition of g' and because x does not occur in u
$\boxed{4}$ by the definition of $\overline{\mathrm{val}}_g^{\alpha_x^v}(u\ F)$
$\boxed{5}$ by the semantics of $\forall x\ u\ F$
$\boxed{6}$ because $\overline{\mathrm{val}}_g^\alpha(I) = 1$ implies $g = g'$
$\boxed{7}$ by the definition of g' and because $\overline{\mathrm{val}}_g^\alpha(u\ F) = 1$

11.6 Function and Predicate Symbols

The proofs of local strong soundness for all states for the rules for function and predicate symbols are simple. In particular for axioms it means that the conclusion of the rule holds in all states. This can be directly verified.

Therefore the local strong soundness for all $g \in \mathbf{G}$ is only proven for some examples.

Non-Rigid Function Symbols The rules for non-rigid function symbols are strongly locally sound because they express precisely the restrictions on states that are imposed in the definition of the standard structure.

The rules (R 59) to (R 63) express that the primary non-rigid function symbols with non-zero arity are only defined for finitely many arguments. And the rule (R 64) applies the definitions of the definable non-rigid function symbols.

Send Position for Sorted Sending For rule (R 118) and arbitrary g and α two cases are distinguished:

- If q, t or i is undefined, the premise and the conclusion are interpreted as 1.
- Otherwise most formulas drop out. Let $\overline{\mathrm{val}}_g^\alpha(i) = n$. Then $\overline{\mathrm{val}}_g^\alpha(G) = 1$ is equivalent to $n \in M$ where M is as in the definition of the semantics of $sendpos^S$, and $\overline{\mathrm{val}}_g^\alpha(F \wedge G) = 1$ is equivalent to $n = \min\ M$. Therefore $\overline{\mathrm{val}}_g^\alpha(F \wedge G) = \overline{\mathrm{val}}_g^\alpha(sendpos^S(q,t) = i)$.

Receive Position for Random Access For rule (R 121) the proof is very similar to the above proof. For arbitrary g and α two cases are distinguished:

- If t_1, ..., t_r, q or i is undefined, the premise and the conclusion are interpreted as 1.
- Otherwise most formulas drop out. Let $\overline{\mathrm{val}}_g^\alpha(i) = n$. Then $\overline{\mathrm{val}}_g^\alpha(G) = 1$ is equivalent to $n \in M$ where M is as in the definition of the semantics of $recpos^{\sigma,S}$, and $\overline{\mathrm{val}}_g^\alpha(F \wedge G) = 1$ is equivalent to $n = \min\ M$. Therefore $\overline{\mathrm{val}}_g^\alpha(F \wedge G) = \overline{\mathrm{val}}_g^\alpha(recpos^{\sigma,S}(q, t_1, \ldots, t_r) = i)$.

For rule (R 122) the soundness result follows because $\overline{\mathrm{val}}_g^\alpha(H) = 1$ is equivalent to $M = \emptyset$ where g, α and M are as above.

Equality of Queues For rule (R 129) and arbitrary g and α two cases are distinguished:

- $\overline{\mathrm{val}_g^\alpha}(q_1) = \bot$ or $\overline{\mathrm{val}_g^\alpha}(q_1) = (m, S', q)$ for $S' \neq S$: $\overline{\mathrm{val}_g^\alpha}(sor^S(q_1)) = 0$ and the premise and the conclusion are interpreted as 1.
- Otherwise: Let $\overline{\mathrm{val}_g^\alpha}(q_1) = (m, S, q)$, then the formulas $sor^S(q_1)$ drop out and the equality of $\overline{\mathrm{val}_g^\alpha}(q_1)$ and $\overline{\mathrm{val}_g^\alpha}(q_2)$ is equivalent to $\overline{\mathrm{val}_g^\alpha}(q_2) = (m, S, q)$. This is what the formulas in the premise express.

Pattern Matching For rule (R 143) and arbitrary g and α two cases are distinguished:

- If a is undefined, the premise and the conclusion are interpreted as 0.
- Otherwise $\left(\overline{\mathrm{val}_g^\alpha}(a), \overline{\mathrm{val}_g^\alpha}(b_1), \ldots, \overline{\mathrm{val}_g^\alpha}(b_r)\right) \in \mathrm{val}_g(match^{\sigma,S})$ holds if and only if $(\overline{\mathrm{val}_g^\alpha}(a))_{i_j} = \overline{\mathrm{val}_g^\alpha}(elem^S_{i_j}(a)) = \overline{\mathrm{val}_g^\alpha}(b_j)$ for $j \in [1; r]$, which is what the premise expresses.

12 Completeness

Proof Sketch In this section the completeness[62] of **C** is proven. The proof consists of two steps: In the first step the completeness for first-order logic is proven. In the second step the completeness for all sequents (relative to the completeness for first-order logic) is proven.

The proof of first-order completeness is trivial because the oracle is defined such that first-order completeness holds. The classical completeness result for first-order predicate logic (see for example [11]) cannot be used because not all first-order structures are considered: The universes are fixed and some of the function and predicate symbols have fixed interpretations (see also section 13.3.4).

The proof of relative completeness relies on four lemmas.

The first two lemmas express common properties of all logics that contain the first-order logic of the natural numbers. The results are elementary and are not proven here. Lemma 12 formalizes that the first-order fragment of DLTP is sufficiently expressive to encode the semantics of all first-order formulas in itself (This so-called Gödelization was introduced in [12].). Lemma 13 formalizes that the first-order fragment of DLTP is sufficiently expressive to describe all recursive sets.

The remaining two lemmas state properties of the modal operators. Lemma 14 states that the semantics of a formula $M(\pi)F$ can be reduced to arbitrarily long, but finite left segments of the traces in the semantics of π. That means that the statement "for all/any traces of π and for all/any (final) states in this trace: F holds" which is used to define the semantics of modal operators can be replaced with "for all/any natural numbers n and for all/any traces of π and for all/any (final) states among the first n states in this trace: F holds".

Then Lemma 15 states that the statement "for all/any traces of π and for all/any (final) states among the first n states in this trace: F holds" for a first-order formula F can be expressed with a first-order formula G. The structure of G depends on M, π and n; in particular n is not a free variable of G.

Theorem 2 uses these four lemmas to constructively show the existence of a formula that makes the Gödelization rule (R 45) locally strongly sound for all states. This formula is a quantification over the Gödel numbers relativized to the Gödel numbers of the formulas G constructed in Lemma 15 for fixed M, π and F.

Using this result the completeness of **C** is proven in Theorem 3.

Other Results The third section gives other completeness results that are less powerful, but more effective than the rule (R 45) or effective in the first place as opposed to the rule (R 58).

12.1 Completeness for First-Order Logic

Lemma 11. **C** *is complete for sequents of first-order formulas.*

Proof. This follows immediately from rule (R 58) (oracle). □

[62] In this section completeness always means completeness for the standard structure and derivation always means derivation in **C**.

12.2 Relative Completeness

See the introduction to this section for a sketch of the proof.

Lemma 12. *There are*

- *an effectively computable injective mapping* $\Gamma : Form \to \mathbb{N}$
- *for every finite* $V \subseteq LV$ *an effectively computable first-order formula* Φ^V *with* $FV(\Phi^V) = V \cup \{y^V\}$ *where* $y^V \in LV(I)$ *is the logical integer variable with the lowest index that is not an element of* V

such that for all $g \in \mathbf{G}$*, all first-order formulas* F *with* $FV(F) = V$ *and all assignments* α *with* $\alpha(y^V) = \Gamma(F)$*:*

$$\overline{\mathrm{val}}_g^\alpha(\Phi^V) = \overline{\mathrm{val}}_g^\alpha(F).$$

Proof. This follows from Gödel's work (see [12]). □

Lemma 13. *For every recursive set* $R \subseteq \mathbb{N}$ *there is an effectively computable first-order formula* F_R *with* $FV(F_R) = \{x_1^I\}$ *such that for all* $g \in \mathbf{G}$ *and all assignments* α*:*

$$\overline{\mathrm{val}}_g^\alpha(F_R) = 1 \text{ if and only if } \alpha(x_1^I) \in R.$$

Proof. This is a fundamental result of computability theory (see for example [8]). □

Definition 53. *For* $n \in \mathbb{N}^*$ left segments *of the length* n *are defined by: For a trace* t *let*

$$t^n = \mathrm{Remove}(t, \{k \in \mathbb{N} | k > n\})$$

and for a set of traces T *let* $T^n = \{t^n | t \in T\}$.

Since the indeterminism in the definition of the semantics of programs is restricted to finitely many choices in each step (see Lemma 1), for all programs π, all $g \in \mathbf{G}$ and all $n \in \mathbb{N}^*$ the set $\overline{\mathrm{val}}_g(\pi)^n$ is finite.

Lemma 14. *For all* $M \in Oper$, $\pi \in Prog$, $F \in Form$, $g \in \mathbf{G}$ *and all assignments* α*:*

$$\overline{\mathrm{val}}_g^\alpha(M(\pi)F) = \left.\begin{cases} \inf\limits_{n\in\mathbb{N}^*} & \text{if } M \in \{[], [[]], \langle[]\rangle\} \\ \sup\limits_{n\in\mathbb{N}^*} & \text{if } M \in \{\langle\rangle, \langle\langle\rangle\rangle, [\langle\rangle]\} \end{cases}\right\} \mathrm{SemModal}(M, \overline{\mathrm{val}}_g(\pi)^n, \alpha, F)$$

Proof.

- The cases where $M \in \{[], [[]], \langle\rangle, \langle\langle\rangle\rangle\}$ are clear.
- Let $M = [\langle\rangle]$:
  - If $\sup\limits_{n\in\mathbb{N}^*} \mathrm{SemModal}(M, \overline{\mathrm{val}}_g(\pi)^n, \alpha, F) = 1$ then clearly $\overline{\mathrm{val}}_g^\alpha(M(\pi)F) = 1$.
  - For the opposite direction assume $\sup\limits_{n\in\mathbb{N}^*} \mathrm{SemModal}(M, \overline{\mathrm{val}}_g(\pi)^n, \alpha, F) = 0$. Let $T = \overline{\mathrm{val}}_g(\pi)$, and for $i \in \mathbb{N}^*$ let a trace $s \in T$ such that $\overline{\mathrm{val}}_{s_j}^\alpha(F) = 0$ for all $j \in \big[1; \min\{i, \mathrm{Length}(s)\}\big]$ be called an i-trace. Then the assumption implies the existence of an i-trace for each $i \in \mathbb{N}^*$. Then the axiom of choice implies the existence of a non-empty set $S = \{s(i) | i \in \mathbb{N}^*\} \subseteq T$ such that for all $i \in \mathbb{N}^*$ $s(i)$ is an i-trace.

    S can be assumed to be an infinite set of infinite traces because otherwise one of its elements would be an i-trace for arbitrarily large i which would immediately yield $\overline{\mathrm{val}}_g^\alpha(M(\pi)F) = 0$.

    For all $i \in \mathbb{N}^*$ a set S_i is defined by induction on i:

    * If $i = 1$: $S_i = S$
    * If $i > 1$: $s^i = s'^i$ defines an equivalence relation on S_{i-1}. Because T^i is finite, this equivalence relation has only finitely many equivalence classes. Since by construction S_{i-1} is infinite there must be an infinite equivalence class. Let S_i be such an equivalence class.

    The axiom of choice is needed to define S_i for arbitrary $i \in \mathbb{N}^*$.

    S contains j-traces for arbitrarily large j. And by induction the same holds for S_i for $i \in \mathbb{N}^*$: firstly $S_1 = S$; and secondly, S_{i+1} is an infinite subset of S_i for all $i \in \mathbb{N}^*$.

    Then by induction on i a trace $t = (t_i)_{i\in\mathbb{N}^*}$ is constructed:

  - If $i = 1$: $t_i = g$. $\overline{\text{val}}_g^\alpha(F) = 0$ holds because (g) must be a 1-trace.
  - If $i > 1$: Let $s \in S_i$. Since S_i contains j-traces for arbitrarily large j and by the definition of S_i all elements of S_i agree in the first i-states, s must be an i-trace and therefore $\overline{\text{val}}_{s_i}^\alpha(F) = 0$. Let $t_i = s_i$. Then $t^i = s^i$ holds.

  The axiom of choice is needed to define t_i for arbitrary $i \in \mathbb{N}^*$.

  Then $\overline{\text{val}}_{t_i}^\alpha(F) = 0$ holds for all $i \in \mathbb{N}^*$. And by the definition of the semantics of programs $t \in T$ holds because for every $i \in \mathbb{N}^*$ there is an $s \in S_i \subseteq T$ such that $t^i = s^i$.[63] And therefore $\overline{\text{val}}_g^\alpha(M(\pi)F) = 0$.

- The case $M = \langle[]\rangle$ follows immediately from the case $M = [\langle\rangle]$ by negating both sides of the equality.

□

Lemma 15. *For all $M \in Oper$, $\pi \in Prog$, first-order formulas F and $n \in \mathbb{N}^*$ there is an effectively computable first-order formula $\phi_n^{M,\pi,F}$ with $FV(\phi_n^{M,\pi,F}) \subseteq FV(F)$ such that for all $g \in \mathbf{G}$ and all assignments α:*

$$\overline{\text{val}}_g^\alpha(\phi_n^{M,\pi,F}) = \text{SemModal}(M, \overline{\text{val}}_g(\pi)^n, \alpha, F).$$

Proof. A formula is called dynamic if it contains a modal operator. It is called untagged if the program in this modal operator (There is only one in the considered cases.) has the empty tag and tagged otherwise. If the tag is of the form i for $i \in \mathbb{N}$ a tagged formula is called asynchronously tagged, otherwise synchronously tagged. A formula that is in the antecedent or succedent of a leaf of the tree $\mathbf{T}$, which is constructed during the following algorithm, is called open.

The Algorithm In order to define the order of rule applications all dynamic formulas that occur in any sequent of the following derivation are labelled with a natural number. After a rule application the introduced dynamic formulas are labelled in the following way.

Labelling: All dynamic formulas that are introduced by the rule application are labelled with the lowest natural numbers that are not yet used as labels; if there is more than one formula to be labelled the order of the labelling is the left-to-right order in which they occur in the definition of the rule. The labels of dynamic formulas in the context of the rule application are not changed.

$\phi_n^{M,\pi,F}$ is constructed by the following derivation algorithm:[64]

1. Input $n \in \mathbb{N}^*$. Let $\vdash M(\pi)F$ be the root of a tree $\mathbf{T}$. Label $M(\pi)F$ with 0. Set $s = 1$.
2. Apply a scheduling rule to the dynamic open formula with the lowest label to which such a rule can be applied and do the labelling. Repeat this step.

   If no such formula exists, go to the next step.
3. Apply an unwinding rule to the dynamic open formula with the lowest label to which such a rule can be applied and do the labelling. If two unwinding rules are applicable to a synchronously tagged formula apply the rule for the case $k = i$. Repeat this step.

   If no such formula exists, go to the next step.
4. Apply a symbolic execution rule to the dynamic open formula with the lowest label to which such a rule can be applied and do the labelling. Repeat this step.

   If no such formula exists, set $s = s + 1$. If $s < n$ go to step 2 and otherwise go to the next step.
5. Apply rule (R 25) to the dynamic open formula with the lowest label to which the rule can be applied. Repeat this step.

   If no such formula exists, go to the next step.
6. Replace all dynamic open formulas with T where T is the timeout condition defined in the statement of rule (R 27).
7. Apply the rule (R 55) wherever possible.

[63] t itself does not need to be an element of S.

[64] The idea of this algorithm is simple: The application of rules to the formulas with the lowest labels and the labelling of the introduced formulas result in a breadth-first search through the tree of traces. The rules are applied such that only $n-1$ commands are executed which means that the left segments of length n of the traces are searched. Then a timeout is enforced.

8. For each leaf $\Gamma \vdash \Delta$ of **T** form the formula $\bigwedge_{G\in\Gamma} G \to \bigvee_{G\in\Delta} G$, and output the conjunction of these formulas.

In this algorithm it is assumed that instead of applying the rules (R 32) and (R 34) the derived rules are applied that arise by additionally applying the rules (R 49) to the formula $U\ (\mathit{Else} \wedge N)$ and then (R 10) to the formula $U\ \mathit{Else} \wedge U\ N$. Thus the modal operators are always in top-level position which is necessary to apply the other rules.

Termination The termination of the loop over the steps 2 to 4 follows because it is repeated precisely $n-1$ times. Then the termination of the algorithm follows from the termination of the single steps. Also for every step the well-definedness must be shown. Both is done in the following.

1. It can easily be shown that if one of the steps 2 to 4 terminates it preserves the following invariant: The tree **T** is finite, and in particular there are only finitely many open dynamic formulas; all dynamic formulas are labelled with different natural numbers.

2. Step 1 is well-defined and implies the invariant.

3. Inspecting the applicability conditions of the scheduling rules (the rules (R 25) to (R 30)) shows that an untagged dynamic formula satisfies precisely one of them, which shows the unique existence of an applicable scheduling rule. This and the invariant imply well-definedness of step 2.

   The application of one of the rules (R 25) to (R 28) decreases the number of untagged open dynamic formulas by one. The application of one of the rules (R 29) or (R 30) leaves this number unchanged, but introduces an untagged open dynamic formula to which neither of the two rules can be applied. That and the invariant imply the termination of step 2 by induction on the number of untagged open dynamic formulas.

4. Inspecting the applicability conditions of the asynchronous unwinding rules (the rules (R 31) to (R 35)) and the asynchronous symbolic execution rules (the instances of the rule schemes (R 41) and (R 42)) shows that an asynchronously tagged dynamic formula satisfies precisely one of them, which shows the unique existence of an applicable unwinding or symbolic execution rule.

   Inspecting the applicability conditions of the synchronous unwinding rules (the rules (R 36) to (R 40)) and the synchronous symbolic execution rules (the instances of the rule schemes (R 43) and (R 44)) shows that a synchronously tagged dynamic formula satisfies precisely one of them, but that there may be two possibilities to choose the value of k from $\{i, j\}$ if this rule is an unwinding rule. The additional condition that $k = i$ is preferred resolves this ambiguity, which shows the existence of a uniquely determined applicable unwinding or symbolic execution rule.

   That and the invariant imply well-definedness of step 3 and, if step 3 has terminated, of step 4.

5. The application of an unwinding rule to a dynamic formula G introduces finitely many dynamic formulas that contain programs with strictly smaller depth of nesting of composed commands (The depth of those processes that are indexed by the tags is used. A formal definition of this depth is given in the proof of Lemma 1.) than G. And this depth is finite for all tagged programs.

   That and the invariant imply the termination of step 3 by induction on the number of tagged dynamic open formulas that contain programs with non-zero depth of nesting and on this depth.

6. The application of a symbolic execution rule decreases the number of tagged dynamic open formulas by one. That and the invariant imply the termination of step 4 by induction on the number of tagged open dynamic formulas.

7. The steps 5, 6, 7 and 8 terminate by the invariant. And the order of rule applications in step 7 is irrelevant because the rule (R 55) replaces one formula with another one.

Partial Correctness Then only partial correctness of the algorithm remains to be proven. Let $g \in \mathbf{G}$ and an assignment α be arbitrary.

If the step 6 of the algorithm were omitted, its output G would satisfy $\overline{\mathrm{val}}_g^{\alpha}(G) = \mathrm{SemModal}(M, \overline{\mathrm{val}}_g(\pi), \alpha, F)$ because for all used rules the soundness criterion of Lemma 4 holds. This is established in the soundness proofs.

If the remaining program is not empty and a timeout occurs, i. e. Ex as in the statement of rule (R 27) and (R 28) is interpreted as 0, applying one of these rules (preceded by the application of the rules (R 29) or (R 30) if a process has exclusive control) reduces to replacing a dynamic formula with T. This is precisely what step

6 does. And because after step 5 no dynamic open formula contains the empty program step 6 corresponds to the application of a locally strongly sound rule if a timeout is enforced.

Since all dynamic formulas that are open before the execution of step 6 have arisen from $M(\pi)F$ by applying $n-1$ symbolic execution rules (and other rules) the execution of step 6 corresponds to enforcing a timeout after the execution of $n-1$ commands (or pairs of commands in the synchronous case) unless the program has already terminated. Therefore the semantics of the computed formula is $\text{SemModal}(M, V, \alpha, F)$ where V arises from $\overline{\text{val}}_g(\pi)$ by cutting off all traces with more than n elements after n elements and labelling the n-th element with $\underline{\text{timeout}}$. By the definition of Fin and $IntMed$ the $\underline{\text{timeout}}$ label is irrelevant, and $\text{SemModal}(M, V, \alpha, F)$ is the same as $\text{SemModal}(M, \overline{\text{val}}_g(\pi)^n, \alpha, F)$.

After step 6 there are no modal operators in open formulas anymore. Therefore all sequences of updates are eliminated in step 7 and the output of the algorithm is first-order and contains at most the free variables of F. □

The following result implies that the rule (R 45) is locally strongly sound for all $g \in \mathbf{G}$.

Theorem 2. *For all $M \in Oper$, $\pi \in Prog$ and first-order formulas F with $FV(F) = V$ there is an effectively computable first-order formula $\phi^{M,\pi,F}$ such that for all $g \in \mathbf{G}$ and all assignments α*

$$\overline{\text{val}}^{\alpha}_g(M(\pi)F) = \overline{\text{val}}^{\alpha}_g(\phi^{M,\pi,F})$$

Proof. The set $S = \left\{\Gamma\big(\phi^{M,\pi,F}_n \wedge \bigwedge_{i=1}^{n} true\big) | n \in \mathbb{N}^*\right\} \subseteq \mathbb{N}$ is recursive: A decision algorithm for the problem $m \in S$ counts the number of occurrences of $true$ in $\Gamma^{-1}(m)$, say N, and computes $\Gamma\big(\phi^{M,\pi,F}_n \wedge \bigwedge_{i=1}^{n} true\big)$ for $n \in [1; N]$. If m is found it outputs is "yes" and otherwise "no".

Let $\psi^{M,\pi,F} = \{y^V / x^l_1\} F_S$ where F_S is the formula constructed from S in Lemma 13.

Then the formula

$$\phi^{M,\pi,F} = \begin{cases} \forall y^V (\psi^{M,\pi,F} \rightarrow \Phi^V) & \text{if } M \in \{[], [[]], \langle[]\rangle\} \\ \exists y^V (\psi^{M,\pi,F} \wedge \Phi^V) & \text{if } M \in \{\langle\rangle, \langle\langle\rangle\rangle, [\langle\rangle]\} \end{cases}$$

is effectively computable from M, π and F and first-order.

And for all $g \in \mathbf{G}$ and all assignments α

$$\overline{\text{val}}^{\alpha}_g(M(\pi)F) \overset{\boxed{1}}{=} \left\{\begin{array}{ll} \inf\limits_{n\in\mathbb{N}^*} & \text{if } M \in \{[], [[]], \langle[]\rangle\} \\ \sup\limits_{n\in\mathbb{N}^*} & \text{if } M \in \{\langle\rangle, \langle\langle\rangle\rangle, [\langle\rangle]\} \end{array}\right\} \text{SemModal}(M, \overline{\text{val}}_g(\pi)^n, \alpha, F)$$

$$\overset{\boxed{2}}{=} \left\{\begin{array}{ll} \inf\limits_{n\in\mathbb{N}^*} & \text{if } M \in \{[], [[]], \langle[]\rangle\} \\ \sup\limits_{n\in\mathbb{N}^*} & \text{if } M \in \{\langle\rangle, \langle\langle\rangle\rangle, [\langle\rangle]\} \end{array}\right\} \overline{\text{val}}^{\alpha}_g(\phi^{M,\pi,F}_n)$$

$$\overset{\boxed{3}}{=} \left\{\begin{array}{ll} \inf\limits_{n\in S} & \text{if } M \in \{[], [[]], \langle[]\rangle\} \\ \sup\limits_{n\in S} & \text{if } M \in \{\langle\rangle, \langle\langle\rangle\rangle, [\langle\rangle]\} \end{array}\right\} \overline{\text{val}}^{\alpha}_g(\Gamma^{-1}(n))$$

$$\overset{\boxed{4}}{=} \left\{\begin{array}{ll} \inf\limits_{n\in S} & \text{if } M \in \{[], [[]], \langle[]\rangle\} \\ \sup\limits_{n\in S} & \text{if } M \in \{\langle\rangle, \langle\langle\rangle\rangle, [\langle\rangle]\} \end{array}\right\} \overline{\text{val}}^{\alpha^n_{y^V}}_g(\Phi^V)$$

$$\overset{\boxed{5}}{=} \left\{\begin{array}{ll} \overline{\text{val}}^{\alpha}_g(\forall y^V (\psi^{M,\pi,F} \rightarrow \Phi^V)) & \text{if } M \in \{[], [[]], \langle[]\rangle\} \\ \overline{\text{val}}^{\alpha}_g(\exists y^V (\psi^{M,\pi,F} \wedge \Phi^V)) & \text{if } M \in \{\langle\rangle, \langle\langle\rangle\rangle, [\langle\rangle]\} \end{array}\right\} \overset{\boxed{6}}{=} \overline{\text{val}}^{\alpha}_g(\phi^{M,\pi,F})$$

□

$\boxed{1}$ by Lemma 14

$\boxed{2}$ by Lemma 15

$\boxed{3}$ by the definition of S and because $\wedge \bigwedge_{i=1}^{n} true$ does not change the semantics

$\boxed{4}$ by Lemma 12 because $\Gamma(\Gamma^{-1}(n)) = n$

$\boxed{5}$ relativization of $Qy^V\ \Phi^V$ for $Q \in \{\forall, \exists\}$ to S, using that $S = \left\{n \in \mathbb{N}^* | \overline{\mathrm{val}_g^{\alpha^n_{y^V}}}(\psi^{M,\pi,F}) = 1\right\}$ by Lemma 13

$\boxed{6}$ by the definition of $\phi^{M,\pi,F}$

Theorem 3. **C** *is complete.*

Proof. Let S be a sequent such that $\overline{\mathrm{val}_g}(S) = 1$ for all $g \in \mathbf{G}$.

The derivability of $S = \Gamma \vdash \Delta$ is shown by induction on the number of non-first-order formulas in $\Gamma \cup \Delta$. The induction step uses the proposition given below to reduce $\Gamma \vdash F, \Delta$ to $\Gamma \vdash F', \Delta$ where F' is first-order. $\Gamma \vdash F', \Delta$ holds in all states because $\Gamma \vdash F, \Delta$ does and only strongly sound rules are applied. If F is in the antecedent the corresponding argument is used. The induction basis is given by the rule (R 58).

The mentioned proposition states that: Every sequent $\Gamma \vdash F, \Delta$ where a formula G occurs in F can be reduced to a sequent $\Gamma \vdash F', \Delta$ where F' arises from F be replacing some occurrence of G with a first-order formula G'; and the reduction uses only strongly sound rules.

The proposition is proven by structural induction on the formula G.

- $G \in AtForm$: G is already first-order and F does not need to be changed.
- $G = H_1 \wedge H_2$, $G = \neg H_1$, $G = \forall x\ H_1$ or $G = \exists x\ H_1$ for $x \in LV$: F' is found by replacing H_1 and, if applicable, H_2 using the induction hypothesis.
- $G = u\ H$ for an update u and a formula H: The induction hypothesis allows to replace H by a first-order formula H'. Applying the rule (R 64) as often as possible to occurrences of definable non-rigid function symbols in H' yields a formula $u\ H''$ where H'' is such that the rule (R 55) can be applied. This rule replaces $u\ H''$ with the first-order formula $\sigma_u(H'')$.
- $G = M(\pi)H$ for $M \in Oper$ and $H \in Form$: The induction hypothesis allows to replace H with a first-order formula H'. Then applying rule (R 45) replaces $M(\pi)H'$ with the first-order formula $\phi^{M,\pi,H'}$.

The soundness result shows that all used rules are strongly sound. □

12.3 Other Results

The results in this section show some partial, but effective completeness results. The proofs are only sketched: Most effort is put into showing which rules are used in which situations of the derivations, and the precise proof that the derivations can be found is omitted or shortened.

12.3.1 Direct Elimination of Programs

The idea of the direct approach is to derive a sequent without using the rule (R 45) which will not be present in an implementation. The following lemma gives a sufficient criterion when this is possible.

Theorem 4. *Let U be a sequence of updates, $M \in Oper$, $\pi \in Prog$ and $F \in Form$ be first-order, let $\Gamma, \Delta \subseteq Form$ and let*

$$S = \Gamma \vdash U\ M(\pi)F\ ,\ \Delta$$

If there is an $n \in \mathbb{N}^*$ such that for all $g \in \mathbf{G}$, all assignments α and all $t \in \overline{\mathrm{val}_{g'}}(\pi)$ where $g' = State(g, \alpha, U)$: $\mathrm{Length}(t) \leq n$,[65] *then S can be reduced to a finite set of sequents of the form $\Gamma, A \vdash S, \Delta$ for finite sets of first-order formulas A and S.*

Proof. This follows from the proof of Lemma 15. The algorithm in that proof is applied with input n and some modifications:

[65]Of course that is not a recursive property of programs.

- The root in step 1 is S. U, Γ and Δ do not occur in the root in the algorithm, but adding them does not interfere with the proof.
- Instead of the steps 5 and 6 step 2 is repeated to apply all scheduling rules.
- And instead of step 8 the following step A is added: Apply the rules for first-order logic and for function and predicate symbols to close all proof goals that still contain modal operators.
- All the leafs of **T** that are not axioms are output.

It remains to prove that the new step A in deed closes all proof goals in which modal operators occur. This follows from the prerequisites of the theorem: If there were such a proof goal which cannot be closed, then there would be a state g such that π can be executed in g' such that there is a trace along which $n-1$ commands are executed and neither program termination nor timeout occur, but then the condition on the lengths of the traces would be violated. □

Then a simple induction that uses the rules for first-order logic and updates ((R 49) to (R 54)) and the rule (R 24) shows that **C** without the rule (R 45) allows to reduce every sequent in which only programs occur that satisfy the above condition to sequents of first-order formulas (which can be derived by the rule (R 58)).

The prerequisite on the length of the traces can be weakened considerably if appropriate induction hypotheses and the rule (R 56) are used to handle programs for which n depends on g and α.

12.3.2 Approximation of the Oracle

C is called effectively complete if **C** without the rule (R 58), which will not be present in an implementation, is complete. This section gives an effective completeness result.

A free, rigid and simple term is called a ground term. The semantics of ground terms does not depend on states or assignments. Therefore let $\mathrm{val} = \overline{\mathrm{val}_O^\alpha}$ for arbitrary α.

Lemma 16. *The standard structure is term generated, i. e. for every sort $S \in \Sigma$ and every $u \in U(S)$ there is a ground term $k(u)$ such that $\mathrm{val}(k(u)) = u$. These terms are called canonical.*

Proof. The canonical terms are defined by:

For all sorts $S \in \Sigma$, $k(\bot) = \bot^S$.

And for $u \neq \bot$, $k(u)$ is defined by:

- If $S = I$: $k(u) = \begin{cases} k(u-1)+c_1 & \text{if } u > 1 \\ c_1 & \text{if } u = 1 \\ c_0 & \text{if } u = 0 \\ k(u+1)-c_1 & \text{if } u < 0 \end{cases}$
- If $S = C$: $k(u) = \begin{cases} \mathit{incchan}(k(u-1)) & \text{if } u > 0 \\ \mathit{nil}^C & \text{if } u = 0 \end{cases}$
- If $S \in \Sigma_P \setminus \Sigma_E$: $k((u_1,\dots,u_n)) = \mathit{tuple}^S(k(u_1),\dots,k(u_n))$
- If $S = Q$: $k((m,S,(u_1,\dots,u_n))) = \mathit{insert}^S(\dots \mathit{insert}^S(()^{m,S}, k(u_n), c_1)\dots, k(u_1), c_1)$
- If $S = P$: $k(\mathrm{run}) = \mathit{run}$, $k(\mathrm{fin}) = \mathit{fin}$

The required properties follow by putting in the definition of the semantics. □

All canonical terms are defined except for the terms $\bot^S$ for $S \in \Sigma$, and all canonical terms contain only canonical subterms. For $S \in \Sigma$ and a ground term $t \in \mathit{Terms}(S)$ let $\bar{t} = k(\mathrm{val}(t))$.

Lemma 17. *For $S \in \Sigma$ and $t \in \mathit{Terms}(S)$* **C** *is effectively complete for the sequent $\vdash t = \bar{t}$.*

Proof. This is shown by induction on t. If $t = f(t_1, \dots, t_n)$ for $n \in \mathbb{N}$ the induction hypothesis and the rule (R 21) allow to assume that the t_i for $i \in [1;n]$ are canonical. Then the rules (R 67), (R 86), (R 116), (R 119) and (R 122) allow to assume additionally that $t_i \neq \perp^{S'}$ for $i \in [1;n]$ and $S' \in \Sigma$, because otherwise $\vdash t = \perp^S$ can be derived.

Then the rules for the function symbol f are used to transform t into a canonical term. However, some of these rules can only be applied if other formulas have been derived before. Therefore some preliminary completeness results are necessary and the induction steps for the function symbols have to be shown in a certain order:

1. **C** is effectively complete for the sequents $t = \perp^S \vdash$ and $\vdash t = \perp^S$ for all canonical terms $t \in Terms(S)$ and $S \in \Sigma$.

   Proof: All defined canonical terms contain only canonical subterms and function symbols for which the rule (R 68) can be applied. Then for defined canonical terms the first sequent can be derived, which is shown by induction on t. For undefined canonical terms the second sequent can be derived by rule (R 124).

2. The induction steps for $f \in \{+, -, *\}$ are shown by using the rules (R 72) to (R 80); the result 1 is used. The induction steps for $f \in \{nil^C, incchan\}$ are trivial. The induction step for $elem_i^S$ for $S \in \Sigma_P$ and $i \in [1; \mathrm{Length}(S)]$ is shown by using the rules (R 94) and (R 96) and result 1.

3. **C** is effectively complete for the sequents $sor^S(q) \vdash$ and $\vdash ssr^S(q)$ for $S \in \Sigma_P$ and a canonical term $q \in Terms(Q)$.

   Proof: This is shown by induction on the length of q. The induction basis is given by the rules (R 145) to (R 147). The induction step uses the rules (R 148) and (R 149) and result 1.

4. **C** is effectively complete for the sequents $P(t_1, t_2) \vdash$ and $\vdash P(t_1, t_2)$ for $S \in \Sigma_P$, $P \in \{=^S, \leq^S\}$ and canonical terms $t_1, t_2 \in Terms(S)$.

   Proof:

   a) The case where $P = \leq^S$ is in the antecedent is reduced to the remaining cases by applying the rule (R 137).

   b) The case where $S \in \Sigma_P \setminus \Sigma_E$ is reduced to the remaining cases by using the rules (R 127), (R 128) and (R 142) and result 2 (for $elem_i^S$).

   c) The case $S = I$ where $t_1 \neq \perp^S$ and $t_2 \neq \perp^S$ is proven by using the rules (R 124), (R 132) and (R 138) to (R 140) and the results 1 and 2.

   d) The case $S = C$ where $t_1 \neq \perp^S$ and $t_2 \neq \perp^S$ is proven by using the rules (R 124), (R 132), (R 91), (R 92), (R 141) and (R 134) and the results 1 and 2.

   e) The case $P = =^S$ and $t_2 = \perp^S$ is shown in result 1. If $t_1 = \perp^S$ the same argument applies after using rule (R 125). The case $P = \leq^S$ and $t_1 = \perp^S$ or $t_2 = \perp^S$ is proven by using the rules (R 132) and (R 136).

5. The induction steps for $f \in \{length, cap\}$ are shown by parallel induction on the number of occurrences of the symbols $insert^S$ for $S \in \Sigma$ in t_1. The induction basis is given by the rules (R 98) and (R 101). The induction step uses the rules (R 99) and (R 102) and the results 2 to 4; the induction hypothesis of this induction is needed to transform the terms $length(q)$ and $cap(q)$ that occur in the rules into canonical terms.

6. The induction steps for the remaining rigid function symbols except for the symbols $sendpos^S$ and $recpos^{\sigma,S}$ for $S \in \Sigma_P$ and $\sigma \subseteq [1; \mathrm{Length}(S)]$ are shown with the following rules:

   - $f \in \{\perp^S | S \in \Sigma\} \cup \{()^{m,S} | S \in \Sigma_P, m \in \mathbb{N}\} \cup \{c_0, c_1, run, fin\}$: t is already canonical.
   - $f \in \{c_n | n \in \mathbb{Z} \setminus \{0,1\}\}$: (R 89) and (R 90).
   - $f = /$: (R 81) to (R 86). If the rules (R 82) to (R 85) are used k must be the appropriate canonical term.
   - $f = ini^S$ for $S \in \Sigma_P$: (R 69) to (R 71).
   - $f = tuple^S$ for $S \in \Sigma_P$: If $S \in \Sigma_E$ the rule (R 97) is used, otherwise t is already canonical.
   - $f = remove$: By induction on $\mathrm{val}(t_2) \in \mathbb{Z}$; the fact that all true subterms of t are canonical and the results 2 to 5 are used. For negative numbers and zero the rule (R 104) is used; for 1 the rule (R 107) is used; the induction step is proven directly using the rules (R 104) or with the induction hypothesis and the rule (R 108).

- $f = insert^S$ for $S \in \Sigma_P$: By induction on $\text{val}(t_3) \in \mathbb{Z}$; the fact that all true subterms of t are canonical and the results 2 to 5 are used. For negative numbers and zero the rule (R 109) is used; the induction step is proven directly using the rule (R 109) or with the induction hypothesis and the rule (R 110).
- $f = read^S$ for $S \in \Sigma_P$: By induction on $\text{val}(t_2) \in \mathbb{Z}$; the fact that all true subterms of t are canonical and the results 2 to 5 are used. For negative numbers and zero the rule (R 116) is used; the induction basis is proven with the rule (R 112); the induction step is proven directly using the rule (R 116) or with the induction hypothesis and the rule (R 113).

7. The induction steps for the symbols $sendpos^S$ and $recpos^{\sigma,S}$ for $S \in \Sigma_P$ and $\sigma \subseteq [1;\text{Length}(S)]$ use the rules (R 118) and (R 119) as well as (R 121) and (R 122), respectively, and the results 2 to 5. Some comments are necessary:
   - For the rules (R 118) and (R 121) i must be the appropriate canonical term.
   - The rules (R 121) and (R 122) introduce the predicate symbol $match^{\sigma,S}$. This is eliminated by the rules (R 143) and (R 144).
   - The rules (R 118), (R 121) and (R 122) introduce formulas of the form $\forall x(c_1 \leq x \leq n \rightarrow Q)$ where n and the free subterms of Q are canonical (If necessary result 5 is used.). If necessary these formulas are derived using the rules (R 14) and (R 88) (induction). The induction basis and the induction step for negative integers are simple. During the induction step for positive integers the formula $\{x/k\}Q$ is introduced for those finitely many canonical terms $k \in Terms(I)$ that satisfy the formula $c_1 \leq^I x \leq^I n$. Which instances are necessary can be decided by the result 4.
   - Result 6 (for $read^S$) is used to make all subterms of the instances $\{x/k\}Q$ of the preceding comment canonical. Then they are derived using the results 2 and 4.

$\square$

Theorem 5. *For a predicate symbol $P \in PS$ with signature $S^1 \times \ldots \times S^n$ and ground terms $t_i \in Terms(S^i)$ for $i \in [1;n]$ let $F = P(t_1, \ldots, t_n)$. Then* **C** *is effectively complete for the sequents $\vdash F$ and $F \vdash$.*

Proof. Using Lemma 17 and the rule (R 21) the terms t_i for $i \in [1;n]$ can be assumed to be canonical. Then

1. The case $P \in \{match^{\sigma,S} | S \in \Sigma_P, \sigma \subseteq [1;\text{Length}(S)]\}$ is reduced to the other cases by the rules (R 143) and (R 144).
2. The case $P ==^Q$: If $\text{val}(t_1) \neq \bot$ by result 3 in the proof of Lemma 17 there is an $S' \in \Sigma_P$ such that $\vdash sor^{S'}(t_1)$ can be derived without using rule (R 58). Applying the rule (R 2) with $F = sor^{S'}(t_1)$ and the rule (R 129) or (R 130) reduces this case to the remaining cases; the introduced universally quantified formula can be derived in the same way as in the result 7 in the proof of Lemma 17. If $\text{val}(t_1) = \bot$ then $t_1 = \bot^Q$ because t_1 is canonical; and then, if $t_2 \neq \bot^Q$, the result is shown with the rules (R 125) and (R 68) by induction on t_1 (where the remaining cases are used), or, if $t_2 = \bot^Q$, with the rule (R 124).
3. The case $P ==^P$ is shown by using the rules (R 131) and (R 125).
4. All remaining cases are already shown in the proof of Lemma 17.

$\square$

Then the rules for first-order logic can be used to show the effective completeness of **C** for all first-order formulas without quantifiers and logical variables. If those are allowed no effective completeness result can be established because of Gödel's incompleteness theorem (see [12]).

12.3.3 Characterization of Universes and States

Two further completeness results are important.

Firstly, some of the universes of the standard structure are essentially determined by the axioms. Informally[66], (R 66) implies that $U(P)$ contains precisely those (three) elements that can be given by ground terms. And (R 65) states the same for the composed sorts provided that the elementary sorts have the property. However,

[66]Stating the result formally would require to define isomorphisms of structures.

the same does not hold for the elementary sorts and the sort Q; this is clear because even the natural numbers alone cannot be determined up to isomorphism by first-order rules.

And secondly, the states of the standard structure are determined by the rules of **C** in the following sense:

- Since the universes of the standard structure are term generated, by Theorem 5 the interpretation of all rigid function symbols and all predicate symbols is determined.
- The rule (R 64) is equivalent to the condition 4 in the definition of the states of the standard structure.
- The axioms (R 59) to (R 63) are equivalent to the condition 5 in the definition of the states of the standard structure.

12.3.4 Used Rules

Inspecting the proofs of the completeness results in this section shows that almost all rules of **C** are used to establish the various completeness properties.

The number of rules that are actually needed for completeness is very small: They are (R 64), (R 55), (R 45) and (R 58). Of course all the other rules are not superfluous, but are needed to replace the rules (R 45) and (R 58), which will not be present in an implementation, as often as possible.

The structural rules (R 1) and (R 2) and the rules for first-order logic ((R 4) to (R 23)) are used implicitly or explicitly in many situations.

The proof of Lemma 15 uses (among others) the rules for the direct elimination of programs ((R 25) to (R 44)) and rule (R 55). But here these rules are only used to show the soundness of rule (R 45), which could also be proven otherwise. In the proof of Theorem 4 they are used in derivations. The latter result can be strengthened by additionally using the rules (R 24), (R 49) to (R 54) and (R 56).

The rules (R 3), (R 46) to (R 48) and (R 57) are only included because they simplify some proofs.

The proof of Theorem 5 uses most of the rules for rigid function symbols and predicate symbols ((R 67) to (R 151)). Only a few rules are not used and can in fact be derived from other rules: The rules (R 150) and (R 151) are not needed since *remove* does not occur in canonical terms; the rule (R 87) is not needed because if $i/j = \perp^I$ is to be disproved it is sufficient to prove $i/j = k$ for a defined term k; the rule (R 95) and some rules of section 10.6.4.7 can also be derived. These rules are included because they are intuitively sound whereas their derivation is complicated.

The characterization of universes needs the rules for universes ((R 65) and (R 66)). And the characterization of states needs the rules for non-rigid function symbols ((R 59) to (R 64)).

13 Discussion

13.1 Possible Generalizations

The concept and structure of DLTP and **C** are very general. Many generalizations can be defined without changing the structure of **C**. In particular the rules for scheduling, unwinding and symbolic execution can be kept unchanged in many cases.

13.1.1 Further Integer Data Types and Type Casts

13.1.1.1 Integer Data Types

Normally integer data types of different sizes are used. These can easily be added by extending the standard vocabulary and the standard structure. For $m \in \mathbb{N}^*$:

- New sort symbols: $U^m, I^m \in \Sigma_E$ (unsigned and signed m-bit integer)
- New rigid program function symbols:
  u^m_n, $\mathrm{Sign}(u^m_n) = U^m$, $n \in [0; 2^m - 1]$
  i^m_n, $\mathrm{Sign}(i^m_n) = I^m$, $n \in [-2^{m-1}; 2^{m-1} - 1]$

- Universes for the new sort symbols: $U(U^m) = U(I^m) = \mathbb{Z}/(2^m\mathbb{Z})$
- Semantics of the new function symbols: $\text{val}_g(u^m_n) = \text{val}_g(i^m_n) = n + 2^m\mathbb{Z}$ for applicable n

Then in particular the data types of Promela are given by: byte $= U^8$, bool $=$ bit $= I^1$, short $= I^{16}$, int $= I^{32}$.

Instead of defining operations for all integer data types type casts are used and the arithmetical operations for the sort I are overloaded.

13.1.1.2 Type Casts

Type casts are introduced by another extension of the standard vocabulary: For all $S, S' \in \Sigma_P$ the following rigid program function symbol is introduced: $\mathit{cast}^{S,S'}$, $\text{Sign}(\mathit{cast}^{S,S'}) = S\ S'$.

The following syntactical abbreviation allows to use implicit type casts: Let $f \in FS$, $\text{Sign}(f) = S'^1\ \ldots\ S'^n\ S'$ and $t_i \in Terms(S^i)$ for $S^i \in \Sigma_P$ for $i \in [1;n]$. Then the term $f(\mathit{cast}^{S^1,S'^1}(t_1), \ldots, \mathit{cast}^{S^n,S'^n}(t_n))$ is abbreviated by $f(t_1, \ldots, t_n)$.

In particular all function symbols can be overloaded: The arguments are cast according to the signature of the overloaded symbol, and the result is cast according to the context.

13.1.1.3 Type Casts for Integer Data Types

The standard structure can be supplemented with the semantics of the type casts. Some of the definitions are given by formulas which makes it easier to add the necessary rules for the new sort symbols and the type casts to the calculus **C**:

For $S, S' \in \Sigma_P \setminus \Sigma_E$:

- If $\text{Length}(S) = \text{Length}(S') = n$: $\mathit{cast}^{S,S'}(t) =^{S'} \mathit{tuple}^{S'}\left(\mathit{cast}^{S_1,S'_1}(\mathit{elem}^S_1(t)), \ldots, \mathit{cast}^{S_n,S'_n}(\mathit{elem}^S_n(t))\right)$
- If $\text{Length}(S) \neq \text{Length}(S')$: $\mathit{cast}^{S,S'}(t) =^{S'} \perp^{S'}$

For $S, S' \in \Sigma_E$:

- If $S = S'$: $\mathit{cast}^{S,S'}(t) =^{S'} t$
- Otherwise and $S = C$ or $S' = C$: $\mathit{cast}^{S,S'}(t) =^{S'} \perp^{S'}$[67]
- Otherwise and $S = I$, $S' = U^m$ or $S' = I^m$ for $m \in \mathbb{N}^*$: $\text{val}_g(\mathit{cast}^{S,S'})(a) = a + 2^m\mathbb{Z}$
- Otherwise and $S = U^m$, $S' = I$ for $m \in \mathbb{N}^*$: $\text{val}_g(\mathit{cast}^{S,S'})(A) = a$ where $A \cap [0; 2^m - 1] = \{a\}$
- Otherwise and $S = I^m$, $S' = I$ for $m \in \mathbb{N}^*$: $\text{val}_g(\mathit{cast}^{S,S'})(A) = a$ where $A \cap [-2^{m-1}; 2^{m-1} - 1] = \{a\}$
- Otherwise: $\mathit{cast}^{S,S'}(t) =^{S'} \mathit{cast}^{I,S'}(\mathit{cast}^{S,I}(t))$

13.1.2 Further Elementary Commands

Elementary commands can be easily added to DLTP. In the following the definitions that have to be extended are given along with an example which adds a command that deconstructs an empty queue:

- Structural definition: Commands can be of the type decq $\subseteq$ elementary. And such a command consists of a channel variable.
- New symbols in the alphabet: DECQ
- New productions in the grammar: command ::= decq and decq ::= DECQ(V(C))
- If necessary, adding a new case in the definition of the remaining program functions. For decq the standard case is sufficient.

[67]This is different from Spin ([2]), which allows type casts between the data types C and I^{32}.

- Definition of executability and effect in all structures: For example for the standard structure, $c = \mathrm{DECQ}(C)$ and a command d,

  $$\mathrm{exec}^{\mathrm{A}}(c) = \{g \in \mathbf{G} | \overline{\mathrm{val}_g}(lengthc(C)) = 0\}$$

  $$\mathrm{exec}^{\mathrm{S}}(c,d) = \mathrm{exec}^{\mathrm{S}}(d,c) = \emptyset$$

  and

  $$\mathrm{val}_{\mathrm{eff}^{\mathrm{A}}{}_c(g)} = (cont, \overline{\mathrm{val}_g}(C), \perp) * \mathrm{val}_g$$

The calculus $\mathbf{C}$ can be extended to a calculus for the extension of the standard structure such that all results on $\mathbf{C}$ are inherited by adding cases in the definition of $Exec^A$, $Exec^S$, Eff^A and Eff^S that satisfy the Lemmas 5, 6 and 7. In the example, for $c = \mathrm{DECQ}(C)$ and $d \in$ elementary, this is achieved by:

$$Exec^A(c) = lengthc(C) =^I c_0$$

$$Exec^S(c,d) = Exec^S(d,c) = false$$

$$Eff^A(c) = (cont, C, \perp^Q)$$

and

$$Eff^S(c,d) = Eff^S(d,c) = ()$$

13.1.3 Further Composed Commands

Adding composed commands is a little more difficult since unwinding is necessary. As above all necessary changes are demonstrated with an example: the composed command unless, which is already part of Promela:

- Structural definition: Commands can be of the type unless. And such a command is a pair of two command sequences.
- New symbols in the alphabet: UNLESS
- New productions in the grammar:

  $$\underline{\text{command}} ::= \underline{\text{unless}}$$

  and

  $$\underline{\text{unless}} ::= \{\ \underline{\text{sequence}}\ \}\ \text{UNLESS}\ \{\ \underline{\text{sequence}}\ \}$$

- New cases in the definition of the unwinding functions:

  $$\mathrm{Unwind}^{\mathrm{A}}(c,g,R) = \begin{cases} \{\mathrm{First}(a);\ unless(\mathrm{Rest}(a), s) | a \in \mathrm{Unwind}^{\mathrm{A}}(s', g, R)\} & \text{if } N \\ \mathrm{Unwind}^{\mathrm{A}}(s', g, R) & \text{otherwise} \end{cases}$$

  $$\mathrm{Unwind}^{\mathrm{S}}(c,t,g,R) = \begin{cases} \{(\mathrm{First}(a);\ unless(\mathrm{Rest}(a), s')\ ,\ b) | (a,b) \in \mathrm{Unwind}^{\mathrm{A}}(s,t,g,R)\} & \text{if } N \\ \mathrm{Unwind}^{\mathrm{S}}(s', t, g, R) & \text{otherwise} \end{cases}$$

  $$\mathrm{Unwind}^{\mathrm{S}}(t,c,g,R) = \begin{cases} \{(b\ ,\ \mathrm{First}(a);\ unless(\mathrm{Rest}(a), s')) | (b,a) \in \mathrm{Unwind}^{\mathrm{A}}(t,s,g,R)\} & \text{if } N \\ \mathrm{Unwind}^{\mathrm{S}}(t, s', g, R) & \text{otherwise} \end{cases}$$

  where $c = \{s\}\mathrm{UNLESS}\{s'\}$, N abbreviates "$\mathrm{Unwind}^{\mathrm{A}}(s'_1, g, R) = \emptyset,\ \mathrm{Unwind}^{\mathrm{S}}(s'_1, r, g) = \emptyset$ for $r \in R$" and

  $$unless(a,b) = \begin{cases} \{a\}\ \mathrm{UNLESS}\ \{b\} & \text{if } a \neq () \\ () & \text{otherwise} \end{cases}$$

  In the above definition $\mathrm{Unwind}^{\mathrm{S}}$ now also depends on R. Clearly the call of $\mathrm{Unwind}^{\mathrm{S}}$ in the semantics of programs must be modified analogously. This signature of $\mathrm{Unwind}^{\mathrm{S}}$ is indeed necessary for the general case and it is a coincidence that the composed commands of DLTP do not require it.

The calculus **C** can be extended to a calculus for the extension of the standard structure such that all results on **C** are inherited by:

- Introducing new cases for the extension of executability that satisfy the Lemmas 5 and 6. For the above example this can be done by:

  $$\mathrm{exec}^{\mathrm{A}}(c) = \mathrm{exec}^{\mathrm{A}}(s_1) \cup \mathrm{exec}^{\mathrm{A}}(s'_1)$$

  $$\mathrm{exec}^{\mathrm{S}}(c,d) = \mathrm{exec}^{\mathrm{S}}(s_1,d) \cup \mathrm{exec}^{\mathrm{S}}(s'_1,d)$$

  and

  $$\mathrm{exec}^{\mathrm{S}}(d,c) = \mathrm{exec}^{\mathrm{S}}(d,s_1) \cup \mathrm{exec}^{\mathrm{S}}(d,s'_1)$$

  for $c = \{s\}\mathrm{UNLESS}\{s'\}$ and a command d

- Adding locally strongly sound unwinding rules. This is possible in the above example. The rules are similar to the unwinding rules for atomic sequences, but even more cases are needed depending on the type of $\mathrm{First}(s)$. Also new unwinding rules for atomic sequences that start with an unless command are necessary.

13.1.4 Further Modal Operators

Abstract Modal Operators Abstractly, the set *Oper* of modal operators of a generalization of DLTP is a subalgebra of the direct product of two algebras O and I with a unary self-inverse operation duality, denoted by $'$.

The elements of O express the quantification over the traces and are denoted by the outer brackets of the modal operator. They are mappings from the power set of $\mathbf{B}$ to $\mathbf{B}$. The dual element m' of m is defined by $m'(B) = \neg m\big(\{\neg b | b \in B\}\big)$. In DLTP for example $[] = \inf$ and $\langle\rangle = \sup$ are elements of O; these are dual to each other.

The elements of I express the quantification over the states within the trace and are denoted by the inner brackets of the modal operator. They are mappings from the set of non-empty sequences over $\mathbf{B}^n$ to $\mathbf{B}$ for some $n \in \mathbb{N}^*$. n is called the arity of the induced modal operators. The dual element m' of m is defined by $m'((B_i)_{i\in D}) = \neg m\big((\neg B_i)_{i\in D}\big)$ where $\neg(b_1,\dots,b_n)$ abbreviates $= (\neg b_1,\dots,\neg b_n)$.

In DLTP for example $[]$ and $\langle\rangle$ are elements of I. They have arity 1, again give the infimum or the supremum, respectively, of all the elements in the sequence and are dual to each other. Two further elements with arity 1 are necessary in DLTP: $[^f]$ and $\langle^f\rangle$ give the supremum or the infimum, respectively, of all the last elements (of which there is at most one) of the sequence and are dual to each other. Then the modal operators $[]$ and $\langle\rangle$ are abbreviations of $[[^f]]$ and $\langle\langle^f\rangle\rangle$, respectively. And the modal operators $[\langle^f\rangle]$ and $\langle[^f]\rangle$ are not elements of *Oper*.[68]

Syntactically, for an n-ary modal operator $M = (o,i)$, where o denotes the outer and i the inner brackets, a program π and formulas $F_1,\dots,F_n$: $M(\pi)(F_1,\dots,F_n)$ is a formula.

To define the semantics of this formula for an interpretation function val_g and an assignment α, first let, for a trace $t = (t_i)_{i\in D}$, ${}^{\alpha}t^{F_1,\dots,F_n} = (B_i)_{i\in D}$ where $B_i = \big(\mathrm{val}^{\alpha}_{t_i}(F_j)\big)_{j\in[1;n]}$ for $i \in D$. Then:

$$\overline{\mathrm{val}}^{\alpha}_{g}\big(M(\pi)(F_1,\dots,F_n)\big) = o\big(\{i({}^{\alpha}t^{F_1,\dots,F_n}) | t \in \overline{\mathrm{val}}_g(\pi)\}\big)$$

Extension of the Calculus Extending DLTP normally means to add further elements to the algebra of inner brackets. Then $Oper^U$ and $Oper^E$ keep their meaning as giving those modal operators that have outer brackets $[]$ or $\langle\rangle$, respectively.

For new unary modal operators two steps are necessary to keep the results on the calculus **C**

- One of the two cases of Lemma 14 must be established for the new modal operators.
- Lemma 15 must be established for the new modal operators. This is most easily achieved by giving additional sound symbolic execution rules and an additional sound program termination rule. Changes to the scheduling and unwinding rules are not necessary (except for possibly the treatment of timeouts).

[68]The operator $[\langle^f\pi\rangle]$ is needed to express intuitively that π terminates (in all traces).

It is conjectured that Lemma 14 is in fact a necessary condition for any sound and complete[69] calculus to exist. This conjecture is supported by the observation that it is not possible to introduce a Gödel numbering of the traces because the set of traces in the semantics of a program may be uncountable. For example for any state g of the standard structure $\overline{\mathrm{val}_g}(\pi)$ is an uncountable set of traces[70] if

$$\pi = \mathrm{DO} \ :: Y_1^I = 0 \ :: Y_1^I = 1 \ \mathrm{OD}$$

For new modal operators with higher arities more changes are necessary.

Examples Examples for new inner brackets are:

- the unary modal operators with the dual inner brackets $[^p]$ and $\langle^p\rangle$. These state that a formula is preserved, i. e. it remains true if it becomes true, and are given by

$$[^p]\big((b_i)_{i\in D}\big) = \begin{cases} 1 & \text{if exists } n \in \mathbb{N}^*, \text{ for all } i \in D: \ b_i = 0 \text{ for } i < n \text{ and } b_i = 1 \text{ for } i \geq n \\ 0 & \text{otherwise} \end{cases}$$

- the unary modal operators with the dual inner brackets $[^i]$ and $\langle^i\rangle$. These state that a formula becomes true infinitely often and are given by

$$[^i]\big((b_i)_{i\in D}\big) = \begin{cases} 1 & \text{if } |\{i \in D | b_i = 1\}| = \infty \\ 0 & \text{otherwise} \end{cases}$$

- the binary modal operators with the dual inner brackets $[^u]$ and $\langle^u\rangle$. These state that the second formula is true until the first formula becomes true and are given by

$$\langle^u\rangle\big(((b_i, c_i))_{i\in D}\big) = \begin{cases} 1 & \text{if exists } n \in D: \text{ for all } i \in [1; n-1]: \ c_i = 1 \text{ and } b_n = 1 \\ 0 & \text{otherwise} \end{cases}$$

Preservation For the modal operators $[[^p\cdot]]$, $[\langle^p\cdot\rangle]$, $\langle[^p\cdot]\rangle$ and $\langle\langle^p\cdot\rangle\rangle$ Lemma 14 can presumably be established, namely the case "inf" if $M \in \{[[^p\cdot]], \langle[^p\cdot]\rangle\}$ and the case "sup" if $M \in \{[\langle^p\cdot\rangle], \langle\langle^p\rangle\rangle\}$.

Sound symbolic execution rules can presumably be given. Let τ be a tag that gives the scheduled (asynchronous or synchronous) execution, E be the executability condition, U' be the sequence of updates U with the effect of the execution added, and π' be the remaining program after the execution. Then the proposed new rules are:

- If $M \in \{[[^p]], \langle[^p]\rangle\}$: $\dfrac{}{\vdash U \ M(\epsilon)F}$
- If $M \in \{[\langle^p\rangle], \langle\langle^p\rangle\rangle\}$: $\dfrac{\vdash}{\vdash U \ M(\epsilon)F}$
- $\dfrac{U\ E\ ,\ U\ F\ \vdash\ U'\ [[\pi']]F \ + \ U\ E\ ,\ U\ \neg F\ \vdash\ U'\ [[^p\pi']]F}{\vdash\ U\ [[^p\tau\pi]]F}$
- $\dfrac{U\ E\ ,\ U\ F\ \vdash\ U'\ \langle[\pi']\rangle F \ + \ U\ E\ ,\ U\ \neg F\ \vdash\ U'\ \langle[^p\pi']\rangle F \ + \ U\ \neg E\ \vdash}{\vdash\ U\ \langle[^p\tau\pi]\rangle F}$
- $\dfrac{U\ E\ ,\ U\ \neg F\ \vdash\ U'\ [\langle\pi'\rangle]F \ + \ U\ E\ ,\ U\ F\ \vdash\ U'\ [\langle^p\pi'\rangle]F}{\vdash\ U\ [\langle^p\tau\pi\rangle]F}$
- $\dfrac{U\ E\ ,\ U\ \neg F\ \vdash\ U'\ \langle\langle\pi'\rangle\rangle F \ + \ U\ E\ ,\ U\ F\ \vdash\ U'\ \langle\langle^p\pi'\rangle\rangle F \ + \ U\ \neg E\ \vdash}{\vdash\ U\ \langle\langle^p\tau\pi\rangle\rangle F}$

That implies that these four modal operators can presumably be added to **C** without losing any result.

Infinite Existence Lemma 14 does not hold for the operators with inner brackets $[^i]$ or $\langle^i\rangle$. Therefore a sound and complete calculus for this extension of DLTP is not expected.

[69] Completeness is used here in the sense that the set of rules is denumerable except for some oracle for sequents of first-order formulas.

[70] The set of states of the standard structure, however, is countable.

Until The analogue to Lemma 14 for higher arities can presumably be proven for these modal operators (the case "inf" for square inner brackets and the case "sup" for angular inner brackets). The necessary rules can presumably be given, too. As an example the proposed rules for the operator $[\langle^u\rangle]$ are given where E, U' and π' are as above:

$$\frac{\vdash\ U\ F}{\vdash\ U\ [\langle^u\epsilon\rangle](F,G)}$$

and

$$\frac{U\ E\ \vdash\ U\ G\ ,\ U\ F\quad +\quad U\ E\ \vdash\ U'\ [\langle^u\pi'\rangle](F,G)\ ,\ U\ F}{\vdash\ U\ [\langle^u\tau\pi\rangle](F,G)}$$

13.1.5 More Synchronous Execution

Promela needs synchronous execution only for synchronous transfers. But the general case used in DLTP where any pair of commands can be synchronously executed is actually simpler than the special case needed for Promela. Only in the standard structure the synchronous executability is restricted to synchronous transfers.

Therefore the definitions of DLTP and the rules of **C** hardly need to be changed if synchronous execution is to be generalized.

More Types of Commands More types of commands can be synchronously executed if another structure than the standard structure is used. If analogues to the Lemmas 5, 6 and 7 can be established for this structure all results on **C** can be kept.

Bigger Number of Commands If more than two commands are to be executed synchronously the definition of structures must be changed by introducing executability conditions for the desired number of commands and appropriate effect mappings.

Then there are several less important changes: new tags, new unwinding functions, semantics of programs with the new tags, scheduling rules that appropriately branch, unwinding rules for programs with the new tags.

But essentially all that is needed is to extend the syntactical description of executability and effect in a way such that the analogues to the Lemmas 5, 6 and 7 hold. Then the symbolic execution rules in the abstract form given in the table at the end of section 10.3.5 can be kept unchanged.

13.1.6 Process and Program States

The definition of process states of DLTP, given by the sort P, is essentially a proof of concept. The process states only contain the information which processes are running or have terminated.

Clearly more information may be desirable, for example about a process:

- the type of the process,
- the ID of the parent and children processes of the process,
- the program counter of the process,
- the number of times the current do loop has been executed,

or about the whole program:

- the number of processes, in total or of a given type,
- the ID of the last scheduled process,
- the number of blocked processes.

This information can be easily provided. It is necessary to

- change the universe of the sort P,
- introduce new rigid logical function symbols that access the information stored in the process state,

- introduce new primary non-rigid function symbols that store the information about the whole program,
- include the corresponding updates for *proc* and the new non-rigid function symbols in the effects of run and end commands,
- if necessary, include updates of these symbols in the effect of any command.

All these changes to the effect functions can be syntactically simulated. And the type and the ID of a process are already stored in a syntactically usable form in the end command.

13.2 Strengths

13.2.1 Formal Semantics and Dynamic Logic for Promela

This work provides one of the first formal specifications of a semantics for (a fragment of) Promela. The existing specifications have several problems.

The official specification ([1]) is informal and not complete. Although the source code of the Spin implementation (e. g. [2]) is publicly available and most widely recognized as defining the semantics of Promela it can hardly be called a formal specification. In particular it is practically impossible to use this semantics in a proof. And even then this implementation has the weaknesses pointed out in section 9.

Several formal semantics have been proposed. In [23] labelled transition systems are used to defined the semantics, but the author himself calls his definitions preliminary[71]. Similarly in [18] the output of an algorithm that is part of Spin and that converts a program into a labelled transition system is used to outline the definition of the semantics of the program in a Kripke structure of system states. However the conversion algorithm itself is not given and many syntactical simplifications are made. Only in [7] a logical definition was approached; the possibility of automated theorem proving is indicated as a future possibility, but the work is preliminary[72] and seems not to have been pursued.

All these proposals have a common disadvantage: The states are mathematically very complicated objects that make both learning and using the definitions difficult. The semantics given in this work uses the well-established foundation of modal logic, and defines states as first-order structures (with rigid universes, partially rigid function symbols and rigid predicate symbols) for a given vocabulary. Therefore the definition in this work has better prospects to become accepted and continued.

In particular it allows to define a dynamic logic around Promela, which is the prerequisite of using deductive verification of Promela programs instead of the model-checking approach that has been used so far. For this logic a sound and relatively complete calculus is given so that automated theorem proving for Promela is now only a matter of implementation.

The most important advantages of the deductive approach are that

- the number of processes is not bounded as in Promela,
- preconditions on the states in which the programs are executed can be given whereas the initial state for the execution of a Promela program is fixed.

In general the tools and experience that are available for dynamic logic can now be applied to Promela which will make the language more useful and interesting.

13.2.2 Indeterministic Dynamic Logic

DLTP is the first dynamic logic for an indeterministic multi-process programming language. The specific problems of such dynamic logics have been solved for Promela in a way that is applicable to most other languages.

These problems are:

- dynamic creation of new processes
- indeterministic scheduling

[71]"neither complete nor error free" (p. 20)
[72]"a work in progress, not a polished result" (p. 2)

- indeterministic execution of processes
- synchronous and buffered asynchronous communication between the processes
- global data objects
- atomic execution
- composed commands that cannot be easily translated into elementary commands
- executability constraints on elementary commands

They represent the elementary problems that arise when indeterministic multi-process dynamic logics are to be developed. And the solutions given in DLTP are recommended for all such logics. In particular this holds for

- the usage of unwinding mappings that translate composed commands into elementary ones,
- the treatment of program variable references in programs,
- the three-stage concept in the calculus, consisting of rules for scheduling, unwinding and symbolic execution,
- the usage of updates to encode state transitions, the update substitution lemma and the rules for updates, which give the usage of non-rigid function symbols to store the system state a strong formal foundation,
- the usage of syntactical elements that are only part of the language because they are needed in the calculus (e. g. updates, conditional terms and tags), which may be rejected as not elegant at first, but is strongly recommended for any dynamic logic for an advanced language.

13.2.3 Temporal Operators for Dynamic Logic

The trace semantics of DLTP is used in most advanced dynamic logics because such logics can use temporal operators that are much more expressive.

The calculus **C** is the first calculus for a dynamic logic that contains the four modal operators $[[]]$, $[\langle\rangle]$, $\langle[]\rangle$ and $\langle\langle\rangle\rangle$ in a way that completely recognizes and exploits the symmetries between them.

As the examples in section 13.1.4 show a wide variety of modal operators can be added to DLTP in a generic way. If a sound and complete extension of **C** for these operators can be given, the new rules will most probably look very similar to the rules of **C**. It may even be possible to give algorithms that produce the necessary rules for certain classes of modal operators.

13.2.4 Abstract Rules

The abstract definition of modal operators used in DLTP that uses outer and inner brackets to specify the semantics of a modal operator (see section 13.1.4) has proven to be very effective. In particular it allows to use symmetries between the modal operators when defining the logic and the calculus.

The power of this abstract definition becomes clear when reading the soundness proofs. Instead of proving the soundness of rules for each modal operator only abstract rule schemes are treated in the soundness proof, and the differences between the modal operators are captured in simple lemmas.

Another proof of the potential of this approach is provided by the examples for possible generalizations in section 13.1: The rules of **C**, in particular the symbolic execution rules, do not change in a fundamental way if commands or modal operators are added.

13.3 Weaknesses

13.3.1 Compositional Semantics

The scheduling of Promela is indeterministic, i. e. the execution of the processes is randomly interleaved. Therefore only elementary commands have a well-defined effect on the global state. And it is not possible to give a compositional definition of the semantics for the composed commands. For due to the interleaved execution of the processes the global system state may be arbitrarily changed between two steps by other processes.[73] Therefore the semantics of programs is defined step by step where in each step one elementary command is executed.

[73]This is true even for commands of the type atomic, because the atomic execution is interrupted if a command is not executable.

Only commands within a process can be composed by the semicolon operator. A composition of programs is not defined in Promela and DLTP. Therefore the composition principle of usual dynamic logic that is captured in the tautology $[\pi][\pi']F \leftrightarrow [\pi;\pi']F$ and that allows to reduce the semantics of the program $\pi;\pi'$ to the semantics of π and π' is not applicable to DLTP. There are several possibilities to define the semantics of the program $\pi;\pi'$, but in most cases the composition principle does not hold. It would hold if $\pi;\pi'$ were defined to be the program in which no command from π' is executed before all commands from π have been executed, but such a composition is not interesting.

In general the composition of programs is less interesting a concept if the programs contain parallelly executed processes. Therefore the composition principle is less important for such dynamic logics.

13.3.2 Concatenation of Modal Operators

In temporal logic the modal operators can be concatenated. For example the formula $\Box\Diamond F$ states that F holds infinitely often on any path along the (irreflexive) accessibility relation. And $\Diamond(F \wedge \Diamond G)$ states that on one path first F and then G holds. That is not easily possible in DLTP (and neither in most dynamic logics) because here the modal operators are relative to a time structure that is induced by a program.

In order to express the above properties of the time structure, a new modal operator (or in the notation for abstract modal operators: a new pair of inner brackets) must be introduced for every needed concatenation of the elementary modal operators.

A general solution could be to separate the program and the type of the modal operator when defining formulas.[74] Then for a formula F and a program π the following formulas would be allowed: $\pi : F$, $\Box F$ and $\Diamond F$. π would be interpreted as a time structure relative to which $\Box$ and $\Diamond$ would be interpreted. But it is difficult to define the semantics of these formulas, let alone give calculus rules.

13.3.3 Loop Invariants

Loop invariant rules are used in most dynamic logics to handle loops that do not terminate or that terminate arbitrarily late. Loop invariant rules can be seen as special cases of the induction rule. Finding an appropriate induction hypothesis is essentially the same as finding an appropriate loop invariant. But loop invariant rules are useful because often the invariant is known by the programmer and can be provided in an interactive proof.

The calculus **C** does not contain loop invariant rules. Giving them is not trivial, because other processes are scheduled during the execution of the loop body which may arbitrarily influence the effect the execution of the loop body has on the loop invariant. Also it must be considered that several processes may be in a loop at the same time and that these loops may create new instances of themselves or each other.

Of course it would be possible to give a loop invariant rule for a do command within an atomic command, but even that rule would be complicated because

- the atomicity is only guaranteed if all commands are executable and no synchronous transfers occur, both of which conditions are not syntactically decidable,
- indeterministic choices must be considered,
- the loop is left indeterministically on a break command somewhere in the loop.

If the execution of a do command is independent of the execution of all other processes and vice versa (and possibly some further restrictions) a loop invariant rule can presumably be given, too.

Another possibility may be to calculate backwards from a break command using the executability conditions and the effects of the preceding commands to prove the post-condition.

A rule for the general case can hardly be given. However, it can be argued that only those programs are interesting for which the interaction of the loops in all processes occurs in some restricted way because otherwise the program is useless. It may be possible to give several loop invariant rules for special cases of how this restriction is formalized.

[74] By design this is automatically the case for TLA (see [17]), which is why the described weakness does not apply to TLA.

13.3.4 Oracle

Strong and Weak Oracle Rule $O' = $ (R 58) is defined such that the completeness of $\mathbf{C}$ for sequents of first-order formulas is trivial. There is a weaker version O of that rule that might be used: The rule O is as O', but restricted to sequents that contain only rigid first-order formulas.

It is not clear whether the calculus with O' instead of O is complete. In the following a proposition is given that is equivalent to the completeness.

To state the proposition some definitions are necessary. Let FO abbreviate many-sorted first-order predicate logic. An FO-vocabulary consists of sorts, function and predicate symbols with their signatures. An FO-structure over a given vocabulary maps each sort to a universe and each function or predicate symbol to an interpretation according to the signature.

Let V' be the FO-vocabulary that is induced by the standard vocabulary for DLTP, and let V be as V', but without the non-rigid function symbols. Let S be the FO-structure over the vocabulary V induced by the standard structure, and for any state $g \in \mathbf{G}$ let S^g be the FO-structure over the vocabulary V' induced by g.

An extension of an FO-structure T over V to V' is an FO-structure over V' that has the same universes as T and interprets the function and predicate symbols of V in the same way as T. Then for every state $g \in \mathbf{G}$, S^g is an extension of S to V'. O is the rule scheme that gives all sequents over V that hold in S, and O' is the rule scheme that gives all sequents over V' that hold in all S^g for $g \in \mathbf{G}$.

Let A be the axioms for primary non-rigid function symbols, i. e. the formulas in the succedents of the conclusions of the rules (R 59) to (R 63).

Then the proposition is: Let T be an FO-structure over V such that T is elementarily equivalent to S and let T' be an extension of T to V' that satisfies the formulas in $A \cup \{\neg F\}$ for an FO-formula F over V'. Then there is an extension S' of S to V' that satisfies these formulas, too.

If this proposition does not hold and if O is used instead of O', then F holds in all states, but cannot be derived, and the calculus is not complete.

Equivalence of the Oracles However, the two rule schemes are equivalent in the following sense. The denumerable set of states of the standard structure can be Gödelized and encoded as natural numbers, let the encoding be Γ. And for every FO-formula F a rigid FO-formula F' can be effectively computed that has an additional free variable $x \in LV(I)$ such that F holds in S^g under the assignment α precisely if F' holds in S under the assignment $\alpha_x^{\Gamma(g)}$.

In other words if the oracle O were present it could be used to implement the oracle O'.

Another possibility is to treat the non-rigid function symbols as free second-order variables. Then rules for a second-order calculus can be used to derive sequents with non-rigid function symbols. Using Gödelization a completeness result can presumably be found since the axioms for the non-rigid function symbols precisely correspond to the restriction that only those functions may be assigned to the free second-order variables that are defined for only finitely many arguments.

13.4 Complexity

This section refers solely to the standard structure and to the calculus $\mathbf{C}$. For simplicity only the derivation of the sequent $S = F \vdash M(\pi)G$ where F and G are first-order formulas, M is a modal operator and π is a program is considered.

The rules are applied with the following priorities:

1. Oracle rule
2. Scheduling rules
3. Unwinding rules
4. Symbolic execution rules
5. Gödelization rules

That means that the oracle is used as early as possible to decide whether first-order formulas can be used to close a proof goal immediately. Gödelization rules are not used, because they render the derivation trivial. The remaining rules are used in the same order as in the algorithm in the proof of Lemma 15, i. e. as many tags as possible are introduced and then all tags are eliminated before a new one is introduced, which corresponds to a concurrent execution of all indeterministic possibilities or a breadth-first search through the tree of traces.

For simplicity the cost of the application of a symbolic execution rule is set to 1 and the cost of the application of any other rule and of all other operations is set to 0. Because each application of a symbolic execution rule is preceded by the application of a scheduling rule, setting the cost of the application of a scheduling rule to 1 would less than double the total cost. Because the number of unwinding rules that precede the application of a symbolic execution rule is limited by the maximum depth of nesting of composed commands that occurs in any used process type, setting the cost of the application of an unwinding rule to 1 would only increase the total cost by a constant factor.

Then let $T(n)$ be the cost of the derivation with the termination criterion that only n commands are executed per trace.[75] Then $T(n)$ is the number of commands that are symbolically executed altogether.

A thorough complexity analysis will not be given. Instead several examples are given that show how the verification of very simple sequents can already be intractable.

Let X abbreviate Y_1^I.

State-Dependent Executability Decisions Let $F = \mathit{true}$, H arbitrary, $M = [[]]$ and

```
$\pi =$ DO
      :: $X;\ X := c_0$
      :: $X + c_1;\ X := c_1$
   OD
```

In every state precisely one option of the <u>do</u> command is executable. Therefore $\overline{\mathrm{val}_g}(\pi)$ has only one element for all states g.

But it depends on the state which option is chosen. Therefore the oracle cannot be used to decide executability and even not executable commands have to be symbolically executed. Therefore[76] $T(2n) = 2\sum_{i=1}^{n} 2^i$

On the other hand if $F = X =^I c_0$ or $F = \neg X =^I c_0$ then[77] $T(2n) = 3n$ because the oracle can decide which option is executable. The cost is also linear if an appropriate cut is applied at the beginning of the proof.

This shows that it is important to restrict the set of initial states as much as possible by giving a strong precondition or to provide formulas to instantiate the cut rule with. In particular if the values of all program variables and queues that are read by a program are given in the precondition or are written by the program before they are read, then executability can always be decided.

State-Dependent Termination A special case of the above example arises when the termination of π depends on the state. Let $H = X =^I c_0$, $M = []$ and

```
$\pi =$ DO
      :: $X;\ X := X - c_1$
      :: ELSE; BREAK
   OD
```

S cannot be derived directly because the termination of π depends on the state in which it is run.

If $F = c_0 \leq^I X \leq^I c_N$, for any $N \in \mathbb{N}$, π terminates if executed in any state g that satisfies F and[78]

$$T(2n) = \begin{cases} 2n & \text{if } n < N+1 \\ 2N+3 & \text{if } n \geq N+1 \end{cases}$$

If $F = c_0 \leq^I X$, S can only be derived with the induction rule (see the example in section 10.7.2) because there are initial states that satisfy F for which the final state is arbitrarily far away.

[75] This corresponds to the algorithm in the proof of Lemma 15 with input $n + 1$.

[76] Each option consists of two commands, therefore $2n$ is used as the argument of T.

[77] The factor 3 occurs instead of 2 because even if a command is not executable, one symbolic execution rule has to be applied to close the proof goal.

[78] If X has been decremented to 0, three further symbolic execution rules are necessary: for X, ELSE and BREAK.

General Cost of Selections The above examples have already indicated that selections can lead to exponential cost. More generally, let $F = \mathit{true}$, H arbitrary, $M = []$ and

```
$\pi$ = DO
        :: $c_1$
        ⋮
        :: $c_r$
      OD
```

for $r \in \mathbb{N}^*$ and $c_1, \ldots, c_r \in$ assignment. Assignments are used for simplicity because they have a non-trivial effect on the state and are always executable. Then

$$T(n) = \sum_{i=1}^{n} r^i \in O(r^n).$$

Dynamic Process Instantiation Let $F = \mathit{true}$, H arbitrary, $M = []$ and $\pi = c$ where

```
$c$ = DO
       :: RUN PT
     OD
```

where PT is a process type without parameters and with $\mathit{Body}(\mathrm{PT}) = c$.

Then $T(n) = \sum_{i=1}^{n} i! \geq n!$.

More Complicated Examples More complicated programs can hardly be analyzed. Already the simple process type PT without parameters and with

```
$Body$(PT) = DO
              :: RUN PT
              :: BREAK
              :: $c_1$
              ⋮
              :: $c_r$
             OD
```

for $r \in \mathbb{N}$ and $c_1, \ldots, c_r \in$ assignment leads to hardly solvable recurrence relations for $T(n)$. A numerical computation shows that even for $r = 1$, already $T(25) > 10^{30}$.

Application of Updates For $n \in \mathbb{N}^*$ let $u_n = (\mathit{var}^I, (\mathit{glob}, c_n), \mathit{var}^I(\mathit{glob}, c_n) + c_1)$, $U_n = u_1 \cdot \ldots \cdot u_n$ and $F = \mathit{var}^I(\mathit{glob}, c_1) =^I c_0$. If C_n denotes the number of conditional terms in $\sigma_{U_n}(F)$, then $C_n = 2^{n-1} \in O(2^n)$.

The reason for the above exponential growth of the length of formulas is that

- each update for a function symbol with non-zero arity introduces a conditional term for each occurrence of the same function symbol in F, and
- the terms in the second and the third component of the updates u_n again contain non-rigid function symbols with non-zero arity that are affected by later update substitutions.

Since most updates that are generated by programs are for program variables and use the values of other program variables to define the new value this effect will appear in almost all cases.

Although this worst-case cost cannot be decreased, in most practical situations[79] the cost can be strongly reduced by using early simplification. In the above example the guards of the introduced conditional terms are of the form $\mathit{glob} =^I \mathit{glob} \wedge c_1 =^I c_n$ for some $n \in \mathbb{N}^*$. And with early simplification $\sigma_{U_n}(F)$ can be computed in linear time and constant space[80].

[79] This is the case because the allowed references to program variables are strongly restricted in Definition 11.
[80] Quadratic time and linear space, if the constant symbols $c_2, c_3, \ldots$ have to expressed by c_1.

Redundant Executability Checks In three situations the calculus **C** checks the executability of commands: Scheduling rules need a case distinction depending on whether any command is executable; unwinding rules for selections with else option need to check the executability of the non-else options; and symbolic execution rules perform the actual executability check. This may lead to redundant derivations.

To alleviate this problem the following heuristic can be applied: Rules are applied to the formulas Ex and $Else$ that are introduced by scheduling and unwinding rules, respectively, with low priority. If an executability formula E that has been introduced by a symbolic execution rule has been proven or disproved this is used to simplify all formulas Ex and $Else$ that contain occurrences of E.

It is also possible to change the rules in a way that reduces the redundancy. In particular the executability check can be removed from the symbolic execution rules and instead added to the scheduling rules. For example rule (R 27) could be changed to

$$\frac{\begin{array}{l} \overbrace{U\ Ex_1}^{\boxed{1}} \vdash \overbrace{U\ M(1:\ \pi)F}^{\boxed{2}} + \ldots + U\ Ex_n \vdash U\ M(n:\ \pi)F + \\ U\ Ex_{1,2} \vdash \overbrace{U\ M(1,2:\ \pi)F}^{\boxed{3}} + U\ Ex_{1,3} \vdash U\ M(1,3:\ \pi)F + \ldots + U\ Ex_{n-1,n} \vdash U\ M(n-1,n:\ \pi)F + \\ \overbrace{U\ \neg Ex\ \vdash\ T}^{\boxed{4}} \end{array}}{\vdash U\ M(\pi)F}$$

where $Ex_i = Exec^A(\text{First}(\pi_i))$ and $Ex_{i,j} = Exec^S(\text{First}(\pi_i), \text{First}(\pi_j))$ for $i, j \in [1; n]$ and $i \neq j$.

Conclusion The above examples indicate that the cost of deriving a sequent is mainly dominated by the complexity of the programs that occur in it. This is an inherent property of Promela and any other dynamic logic for Promela will show the same intractable behavior. Therefore the deductive approach cannot be much more complex than the model-checking approach. However, the latter has the advantage that Spin has already been optimized for efficiency whereas the complexity analysis of DLTP has so far only been rudimentary.

Therefore there is one fundamental objective in the application of DLTP: to limit the number of branches in the proof. This can be achieved by various measures:

- writing the programs in a way that minimizes branching,
- providing preconditions from which the non-executability of as many commands as possible can be derived,
- introducing the concept of commutativity of commands: Informally, two elementary commands are commutative if their joint effect is the same no matter in which order they are executed. A typical example for commutative commands are commands of different processes that operate only on the respective local variables. If two commutative commands are eligible for execution only one order must be considered.
- giving additional rules for DLTP that are analogous to the partial order reduction algorithm of Spin,
- introducing priorities for processes or options in order to decrease the number of indeterministic choices,[81]
- introducing further rules that allow to skip parts of the elimination of a program.

The chances of the last measure are weak. But there are some special cases where it might be helpful, for example:

- The rules of section 10.4 already show how rigid parts of a formula can be eliminated from a modality.
- If it is possible to foresee that an update u will not be overwritten by the execution of a program π, then u can already be applied to F in $u\ M(\pi)F$.
- Loop invariant rules (see section 13.3.3) and appropriate induction hypotheses may allow to skip do commands.

13.5 Future Work

Efficiency The most important improvement of DLTP is to add rules or even to change definitions in order to make the derivation more efficient. Some ideas how this can be done are given in section 13.4.

[81] Of course then Promela becomes the (irrelevant) special case where all priorities are equal.

Loop Invariant Rules It is very promising to try to find rules that use loop invariants, at least in special cases. This is discussed in section 13.3.3.

Implementation The rules of **C** are stated in a form that allows a convenient implementation. Most sequent calculus implementations can be easily adapted to DLTP and **C**. The important details are parsing the grammar form of a program into the structural form, and handling updates and conditional terms.

For example the KeY system ([3]) uses taclets (see [4]) to flexibly specify rules of a Java Card calculus. If a Promela parser is added and the syntax of updates is extended, this formalism or a slightly generalized version can be used to describe the rules of **C**.

Rules for Abstract Modal Operators It appears to be a promising task to try to find rules for abstract modal operators as defined in 13.1.4. At least for special cases it should be possible to give an algorithm that constructs the necessary rules from the abstract definition.

The semantics of the outer brackets should be considered in the scheduling and the unwinding rules and the semantics of the inner brackets in the symbolic execution rules.

For the latter it may be interesting to extend the syntax of formulas in the way "For a modal operator M, a program π, a formula F and $n \in \mathbb{N}$: $nM(\pi)F$ is a formula.". Then $n = 0$ reduces to DLTP formulas and $n \neq 0$ offers a way to store the state of an automaton that accepts sequences of truth values. For each state of this automaton one rule is necessary and transitions that lead into accepting states correspond to axioms.[82]

A General Problem In section 13.3.4 the general problem of a completeness result (relative to some oracle) for first-order predicate logic with fixed universes and fixed interpretations for some symbols is encountered. This appears to be an open problem.

Bibliography

[1] Promela Language Reference, Version 3.[83] Available in the ONLINE REFERENCES section at http://spinroot.com (or in G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley, 2003). 1.2.1, 1.2.2, 2, 1.3, 17, 26, 27, 8.2, 44, 13.2.1

[2] Spin, version 4.12 (open source), precompiled binary for Windows PC systems from http://spinroot.com. 1.2.1, 1.3, 26, 27, 40, 9.3.3, 9.3.3, 9.3.5, 9.3.5, 42, 67, 13.2.1

[3] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager and P. H. Schmitt, *The KeY Tool*, Software and System Modeling, 2003. 13.5

[4] B. Beckert, M. Giese, E. Habermalz, R. Hähnle, A. Roth, P. Rümmer and S. Schlager, *Taclets: A New Paradigm for Constructing Interactive Theorem Provers*, Revista de la Real Academia de Ciencias, Serie A. Mat. vol. Falta, pp. 1-36. 13.5

[5] B. Beckert and S. Schlager, *A Sequent Calculus for First-order Dynamic Logic with Trace Modalities*, International Joint Conference on Automated Reasoning, Siena, Italy, pp. 626-641, LNCS 2083, Springer, 2001. 1.3

[6] B. Beckert, *A Dynamic Logic for the formal verification of Java Card programs*, Java Card Workshop, Cannes, France, LNCS 2014, 2001. 1.3

[7] W. R. Bevier, *Towards an Operational Semantics of PROMELA in ACL2*, Spin Workshop, Twente University, Enschede, The Netherlands, 1997. 1.3, 40, 13.2.1

[8] G. S. Boolos, R. C. Jeffrey, *Computability and Logic*, Cambridge University Press, 1974. 12.2

[9] E. M. Clarke, E. A. Emerson and A. P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Programming Languages and Systems 8(2), pp. 244-263, 1986. 1.1, 4.3

[10] S. A. Cook, *Soundness and Completeness of an Axiom System for Program Verification*, SIAM Journal on Computing 7(1), pp. 70-90, 1978. 1.3

[82] In most cases such a finite automaton can only be used for one of two dual modal operators.
[83] The online reference is used in this work. It gives its version inconsistently as 3.0 or 3.3.

[11] J. H. Gallier, *Logic for Computer Science: Foundations of Automated Theorem Proving*, Harper and Row, 1986. 1.3, 11.2, 12

[12] K. Gödel, *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*, Monatsheft für Mathematik und Physik 38, pp. 173-198, 1931. And *Diskussion zur Grundlegung der Mathematik, Erkenntnis 2*, Monatsheft für Mathematik und Physik, pp. 147-148, 1931-32. 1.3, 12, 12.2, 12.3.2

[13] D. Harel, *First-order Dynamic Logic*, LNCS 68, Springer, 1979. 1.1

[14] D. Harel, *Dynamic logic*, in Handbook of Philosophical Logic, volume 2, chapter 10, pp. 497-604, Reidel, 1984. 1.1

[15] D. Harel, D. Kozen and J. Tiuryn, *Dynamic Logic*, MIT Press, 2000. 1.3

[16] G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991. 4.4

[17] L. Lamport, *The Temporal Logic of Actions*, ACM Transactions on Programming Languages and Systems, 16(3), pp. 872-923, 1994. 1.3, 74

[18] V. Natarajan and G. J. Holzmann, *Outline for an operational semantics definition of PROMELA*, technical report, Bell Laboratories, July 1996. 1.3, 13.2.1

[19] R. Parikh, *A decidability result for second order process logic*, 19th IEEE Symposium on Foundation of Computer Science, pp. 177-183, 1978. 1.3

[20] D. Peleg, *Concurrent dynamic logic*, ACM, 34(2), pp. 450-479, 1987. 1.3

[21] V. R. *Pratt, Semantical considerations on Floyd-Hoare logic*, 18th IEEE Symposium on Foundation of Computer Science, pp. 109-121, 1977. 1.1, 1.3

[22] V. R. Pratt, *Process logic: Preliminary report*, ACM Symposium on Principles of Programming Languages, San Antonio, USA, 1979. 1.1, 1.3

[23] C. Weise, *An incremental formal semantics for PROMELA*, Spin Workshop, Twente University, Enschede, The Netherlands, 1997. 1.3, 13.2.1