The Correctness of the PACLIB Kernel

A Case Study in Parallel Program Verification by Temporal Logic*

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
email: `schreine@risc.uni-linz.ac.at`

April 20, 1993

Abstract

We verify the correct implementation of a non-deterministic construct for the delivery of task results in the parallel computer algebra library PACLIB*. First we explain the corresponding part of the* PACLIB *runtime kernel. Then we give an overview on the calculus of temporal logic and extend it for our purposes. We describe how the* PACLIB *kernel is modelled in this formal framework. We formulate several key theorems and auxiliary lemmas that capture the dynamic behavior of the construct and verify their correctness. Based on the experiences of this case study, we describe some general verification strategies.*

*Supported by the FWF grant S5302-PHY "Parallel Symbolic Computation".

Contents

Beware of bugs in the above code;
I've only proved it correct, not tried it.

Donald Knuth

1 Introduction

The verification of parallel programs is a very complicated and tedious task. The more freedom a programming language provides for inter-process communication and synchronization, the more complex the interaction between parallel processes may become, the more error-prone a program is and the more issues a verification has to consider. One of the most difficult cases is that of a program where many processes are tightly coupled by shared memory and synchronized by semaphores or equivalent constructs.

High-level parallel programming languages therefore offer more restricted forms of communication and synchronization that yield less error-prone programs whose correctness may be verified more easily. However, the implementation of a high-level parallel programming construct is based on low-level concepts; its verification is therefore a difficult task in its own. One reason for this is that the correctness of a language construct has to be established for *every* program that applies the construct!

In this paper, we will verify the correct implementation of the non-deterministic function `pacWait` that is used in the parallel computer algebra library PACLIB [HSN+92] for the delivery of task results. PACLIB programs using `pacWait` are quite loosely coupled and their verification is in general rather easy (i.e. the correctness of the parallel program can be essentially reduced to the correctness of the sequential program). However, the implementation of `pacWait` itself demands a complicated interaction of tightly coupled parallel processes. The `pacWait` algorithm is relatively short, but its dynamical behavior makes a verification rather difficult.

There exist several calculi for the formal verification of parallel programs e.g. the axiomatic model by [OG76, OL82], the Unity model [CM88], Millner's CCS [Mil80], the Π-calculus [Mil93], Hoare's CSP [Hoa85] or temporal logic [MP81, Krö87]. The introduction of these calculi usually consists of the proof of their mathematical soundness and completeness and their application to small example programs.

However, techniques that may be useful for the verification of small programs may fail for more realistic examples. The reason for this are not mathematical deficiencies but pragmatic ones, for instance if the complexity of verification is prohibitive. Therefore, realistic examples are necessary not only to evaluate the practical usefulness of a calculus but also to suggest adaptations and modifications that may increase its range of application. Unfortunately, there are only few such examples in literature [Oss83].

The goal of this paper is to present the verification of a parallel program with a rather complex dynamical behavior whose correctness is not immediately clear. We do not start the verification with a nice formal model but first present the "real-world" problem that we actually want to be solved, then show how this problem can be modelled in a formal framework and finally derive the properties of this model.

The next section of this paper contains several "meta-remarks" on the issue of parallel program verification, its usefulness and its limitations. The main components of the case study are then presented in the following sections:

1. the description of `pacWait` and its implementation,
2. the framework of temporal logic,
3. the formal model and the correctness theorem, and
4. the verification itself.

Some general strategies for parallel program verification derived from our case study conclude the paper. The appendix contains the source code of the actual program and the proofs of the lemmas used in the verification.

2 Parallel Program Verification

In this section, we will present our view on the formal verification of parallel programs, its usefulness and its limitations. We will argue that

1. The goal of parallel program verification is to increase the confidence in some properties of an executable parallel program.
2. However, the formal verification process is only able to deal with a reduced model of this program.
3. The adequacy of the model with respect to the interesting properties of the real program cannot be checked by formal methods.
4. Even a formal verification of the model in general extremely complex and only feasible by radical simplifications.
5. Parallel program verification is therefore as error-prone as parallel programming itself and its adequacy cannot be guaranteed.
6. The importance of parallel program verification is that of documenting a parallel program more than of proving it.

The section is rather (maybe too) lengthy and mainly contains the personal options of the author. The hasty reader only interested in the formal part of this paper may therefore skip it and directly go the next section.

2.1 Organization of the Verification

The process of program verification consists in our opinion of the following components:

1. **A Running Program:** There is given a program p that has in all past test runs shown the desired behavior. However, the programmer would like to increase his confidence that the program will show this behavior for all future runs (and all given inputs) and/or convince other people of this fact. For this purpose, informal reasoning in a natural language is by its ambiguity not considered sufficient.
2. **A Formal Framework:** The programmer chases a framework that allows more precisely than natural language to express properties of the program and to reason about these properties. To simplify the reasoning, this framework is a reduced image of the "real world" (i.e. computer hardware, operating system, programming language) neglecting all effects that the programmer considers of minor importance for the aspects of the program he is interested in. By the formal structure of the model, the correctness of the reasoning process can be more easily controlled.
3. **Program Model:** In the chosen framework, the programmer constructs a model $\bar{p}$ of the program and a proposition $c_{\bar{p}}$ that describes the notion of "correctness" in this model. If for each possible "model run" r of $\bar{p}$ the proposition $c_{\bar{p}}(r)$ holds, the program is considered correct. If a single counterexample r' with $\neg c_{\bar{p}}(r')$ can be constructed, the program is considered incorrect. However, the question whether $\bar{p}$ and $c_{\bar{p}}$ adequately describe p and its effects in the "real world" is not a question that can be answered by formal methods.
4. **Correctness Proof:** In the chosen framework, the programmer argues why the property $\forall r(c_{\bar{p}}(r))$ is true. Guided by some proof strategy, he reduces this proposition to other propositions whose correctness can be verified more easily. Ideally, the reasoning process is carried to an end where the propositions and the reasoning becomes so simple that it can be automatically checked by a computer. Practically, the reasoning process is stopped at a level where the correctness of the propositions is considered "obvious" for a human being.

2.2 Program Model and Implementation

The organization of program verification presented above contradicts the common view of the matter concerning the role of p and $\bar{p}$, respectively:

- In the common view, $\bar{p}$ is considered as the *algorithm* and the central point of discussion and p is considered as the *implementation* of $\bar{p}$.
- In our view, $\bar{p}$ is a *model* of p. Actually, we are interested in the properties of the program p and have introduced $\bar{p}'$ to make the domain of discourse tractable.

The main difference between both points of view is the importance that is given to the "mapping" of p to $\bar{p}$ or vice versa, respectively. Many promoters of the common view consider the design of p (if they are at all aware of the difference between p and $\bar{p}$) as a tedious but actually not very interesting task (since p is just the implementation of the $\bar{p}$ whose correctness has already been verified).

This view has its origin in the fact, that formal frameworks for sequential algorithms very well capture the main properties of real programs one is usually interested in. These properties are whether the program terminates and delivers the correct result for any valid input (and how long the computation takes). For an adequate description of these properties, many details of the program may be neglected. Most important, the model needs not contain any notion of *time relations*, since a program run can be easily modelled as a static sequence of variable assignments.

However, in parallel programming the gap between an abstract model and a concrete implementation is often very wide; a lot of textbooks on parallel algorithms use a notation that is far away from the constructs in which parallel programs have to be actually implemented. Many properties of the concrete program p can then not at all be derived from properties of the the abstract model $\bar{p}$, since these properties essentially depend on aspects that are usually decided at the time of the implementation only. If we only construct an abstract model $\bar{p}$ and verify its correctness, it is easily possible that the model will be to simple to capture the essentials of the concrete p.

Of course, before a programmer starts implementing a parallel program, he has designed some abstract algorithm $\bar{p}$ and convinced himself about the correctness of the ideas expressed in $\bar{p}$. However, as explained above, the actual implementation p will usually have a significant feedback also on $\bar{p}$ making the model more realistic and useful. Therefore we think that a prototype implementation p should precede a formal verification of $\bar{p}$. This has also the advantage that by testing p, many "trivial" errors in $\bar{p}$ (like wrong boundary conditions) can be found in an early stage far more easily than by a formal verification.

2.3 The Complexity of Verification

But if $\bar{p}$ is only a simplified model of p, are we then at least able to guarantee the correctness of $\bar{p}$ within the formal framework? Before we discuss the question, we would like to draw attention to the issue of the complexity of program verification.

Verifications of sequential programs are *compositional* in the sense that properties of independent program parts can be independently verified. For instance, in order to prove

$$\{a\}P;Q\{b\}$$

in Hoare's calculus, it suffices to prove for some proposition c

$$\{a\}P\{c\} \ \wedge \ \{c\}Q\{b\}$$

where both proofs can be carried through independently of each other. The verification of a large program can therefore be neatly decomposed into verifications of independent modules; the complexity of the verification reflects the complexity of the program structure.

In the case of parallel programs, the situation is fundamentally different. The proof of

$$\{a\}P||Q\{b\}$$

can not be decomposed into independent proofs on P and Q, since each operation in P may in general have some effect on the execution of Q. Instead, we have to show that if a initially holds, b holds after the execution of the program *for any possible interleaving order* of the program operations.

Of course, if the processes contain sequential modules that do not interact with other processes, these modules may be considered as "atomic" and their correctness can be independently checked by sequential verification methods. However, the complexity of the proof grows in a combinatorial way with the number of interactions and the number of processes involved in these interactions. Therefore, for tightly coupled parallel processes with complex interaction patterns a fully formal verification is in general not feasible. We consider this as a principal limit to formal program verification that can not be overcome no matter which formal framework is applied.

There are two possibilities to cut down the complexity of the verification to a tractable order. The first one is to use parallel programming languages that restrict the interaction between parallel processes to particular patterns for which simple formal frameworks can be constructed (see Subsection 2.5). Even if the parallel program is developed in a language that allows very general forms of interactions, many simplifications can be performed, if these general forms are not really applied. However, then the verification also has to include the proof that the simplifications are really valid, which may be a very complex task in its own.

For parallel programs where the general forms of interaction are actually applied, the only chance to make a verification manageable is to let the human brain serve as a "preprocessor" performing radical simplifications by investigating only the "interesting" cases and by combining several inference steps into one using "higher-order" verification rules. However, the justification of these simplifications itself is then necessarily an informal one and prone to human errors. The persuasion of the proof decreases up to some point where the verification itself becomes less convincing than the fact that the program p has shown the correct behavior for a finite amount of times.

The saying "testing is only able to detect errors, the absence of errors can be only guaranteed by a formal verification" seems therefore questionable. Verifications are not just "right" or "wrong" (this could be only checked by a computer if the verification was fully formal), they are more or less "convincing" for human beings! The whole process of parallel program verification is a steady tradeoff between the two extremes of simplicity and significance, between readable proofs that are not very persuasive and convincing proofs whose details nobody is able to follow.

These arguments are of course also true for the verification of sequential programs or (to some extent) for the proof of any mathematical properties. However, in the case of parallel program verification they become especially striking, since the principal complexity of the verification process itself is much larger.

2.4 The Purpose of Verification

What is then the use of parallel program verification? A fully formal verification is in general not tractable, it is uncertain whether the program model captures all important effects and the proof itself may contain large informal inference steps. A formal program verifications is therefore at least as prone to errors as the process of programming itself.

Actually we consider as the main value of the verification of a parallel program not the aspect of *proving* (in the sense of establishing without doubt) the correctness of a program but the aspect of *documenting* the programmer's knowledge by

1. formal propositions that he believes to capture the essential properties of the program and by

2. (semi-)formal arguments that support the believe in the correctness of these propositions.

By a program verification, the programmer is (by some proof strategy) forced to methodologically think about the program and to formulate his thoughts, his knowledge and his opinions as formal propositions and (semi-)formal proofs. By this process itself, many errors can be captured and the confidence in the correct behavior of a program can be considerably increased. The result of this process, the propositions and the proof, may then serve as a documentation of the program and as a trace of the programmer's thoughts. Moreover, by the formal notation, the propositions can be irrefutably falsified by anybody that is able to construct counterexamples.

2.5 The Role of the Parallel Programming Model

The parallel programming model (respectively language) for implementing p may have some important impact on the verification process. The arguments concerning the complexity of parallel program verification presented in the previous subsections mainly applied to the classical parallel programming models with sequential processes coupled by a global store (shared memory programming) or special communication channels (message passing programming). The more tightly coupled the parallel processes are, the more complex their interaction and therefore their verification becomes.

On the other hand, parallel models based on concepts such as functional programming are much more easy to understand. The only interaction between processes occurs at the time of process creation (when arguments are passed) and process termination (when the result is delivered). For such models, no special parallel verification techniques are required, since here the correctness of the parallel program is a direct consequence of the correctness of the sequential program.

The parallel programming model of PACLIB [HSN+92] is a compromise between both extremes. If only the deterministic construct `pacWaitOne` is applied for communication between processes, the correctness of a PACLIB program directly follows from the correctness of the corresponding SACLIB program where all `pacStart` and `pacWaitOne` constructs are removed. Even if the non-deterministic `pacWait` construct is used, program verification is fairly easy. Only, if processes are coupled by streams using `pacPut` and `pacGet` (or even worse by buffers using `pacRead` and `pacWrite`), the verification of programs becomes extremely complex. The purely functional model of parallelism should therefore be preferred, since the resulting programs are by orders of magnitude far more easy to understand and to verify.

3 The PACLIB Kernel

The PACLIB kernel is a runtime system for shared-memory multiprocessors that can be linked to any C program and provides light-weight concurrency together with a garbage-collected heap.

In this section, we will describe one aspect of the PACLIB kernel whose correctness we are especially interested in: the non-deterministic task delivery scheme of the PACLIB synchronization construct `pacWait`. The contents of this section are essentially taken from [Sch92].

3.1 The Programming Model

The Paclib model of parallelism is basically a *functional* one: A Paclib task receives certain Saclib objects as input arguments and returns a Saclib object as its result. This result may be retrieved later by a handle to this task.

task = `pacStart(`*fun*`,` *args*`)`

creates a new task that executes *fun(args)* and can be referenced via the handle *task*. *task* is a first-order Saclib object i.e. it can passed to other functions or tasks or stored in a Saclib object. The result of this task (which is the value returned by *fun*) may be retrieved (arbitrarily often) by any task that knows *task*.

value = `pacWait(`*tptr*`,` *tasks*`)`

returns the result *value* of one of the *tasks* and writes the handle of the delivering task into the location *tptr*. If all denoted tasks are still active, `pacWait` blocks until one task terminates and then delivers the result of this task.

3.2 Task Creation

There are two separate entities defining a task:

- a *task descriptor* that exists as long as the task is alive.
- a *result descriptor* that exists as long as the task can be referenced.

The reason for this distinction is that the task descriptor is very large but only required as long as the task is running (i.e. its space can be reclaimed on the termination of a task). On the other hand, the result descriptor can be very small but it must exist as long as any other task might want to receive the task result.

The internal interpretation of the *task* handle returned by *pacStart* is that of a *result descriptor* which is a (reference to a) Saclib list with the following elements:

`sem`	`task`	`wait`	`value`

- `sem` is (a pointer to) a semaphore that provides mutual exclusion on the result descriptor.
- `task` is a pointer to the task descriptor of the task that computes the result. If the task has already delivered its result, `task` is the special constant `AVAILABLE`.
- `wait` contains, if `task` $\neq$ `AVAILABLE`, the head of the queue of task descriptors of all tasks blocked on an attempt to receive the result.
- `value` contains, if `task` $\neq$ `AVAILABLE`, the tail of the `wait` queue. If `task` $=$ `AVAILABLE`, it contains the result of the task.

A *task descriptor* is a record that contains the workspace (i.e. the stack) of the task and the following fields (among others):

<table>
<tr><td colspan="2">sem</td></tr>
<tr><td>prev</td><td>next</td></tr>
<tr><td colspan="2">result</td></tr>
<tr><td colspan="2">value</td></tr>
<tr><td colspan="2">wait</td></tr>
<tr><td colspan="2">sub</td></tr>
<tr><td>stack</td><td>stop</td></tr>
</table>

- `sem` is a (pointer to a) semaphore that provides mutual exclusion on the task descriptor.
- `prev` and `next` connect all existing task descriptors into a doubly linked list.
- `result` is a reference to the result descriptor of the task.
- `value` is a location that receives the result of another task if the current task is blocked in a call of `pacWait`.
- `wait` contains, if the task is blocked in a call of `pacWait`, a list of those result descriptors that the current task is waiting for.
- `sub` is a list of result descriptors of all subtasks created by the current task.
- `stack` is an integer that denotes the stack size to be allocated for all tasks created by the current task.
- `stop` is a flag that is set if another task wants to abort the current task.

When a new task is started, a result descriptor and a task descriptor are allocated. The result descriptor is initialized with a new semaphore, a pointer to the task descriptor and an empty task queue. The fields of the task descriptor are initialized with the start address of the function to be executed, the corresponding arguments, a reference to the result descriptor and all the other values required to determine the task. The task descriptor is finally put into the ready queue for execution.

Actually, the new task does not execute the function *fun* provided by the user but a function `pacShell` that receives the address of *fun* and the corresponding arguments. `pacShell` calls *fun* and gets the result of this function call. This result is stored in the `value` field of the result descriptor and all tasks blocked on this descriptor are awaked. The task descriptor is then deallocated and the task terminates. The delivery of task results is described in the next subsection in more detail.

3.3 Task Results

The algorithm that implements `pacWait` works roughly as follows (the numbers in Figure 1 refer to the steps of the algorithm):

1. **Lock:** The semaphore associated to the descriptor t of the current task is locked. This prevents the *tasks* from delivering their result to t before `pacWait` has reestablished a consistent state.
2. **Scan:** All result descriptors in *tasks* are scanned and the result descriptor r of the current task is linked into the waiting queues of these descriptors.
3. **Found:** If a result descriptor in *tasks* already contains a result, r is removed from all waiting queues it has previously been linked into. t is unlocked and the result value is returned.
4. **Sleep:** If none of the results is already available, the result descriptor list *tasks* is stored in the `wait` field of t. t is unlocked and the current task is blocked.
5. **Awake:** After the task has been awaked, the `value` field of t contains the result of one of the *tasks* it waited for. Likewise, the `wait` field now contains the result descriptor of the task that returned the result.

After the `pacShell` has received the result of *fun*, it forwards the value to all tasks waiting on the result descriptor r (see Figure 2):

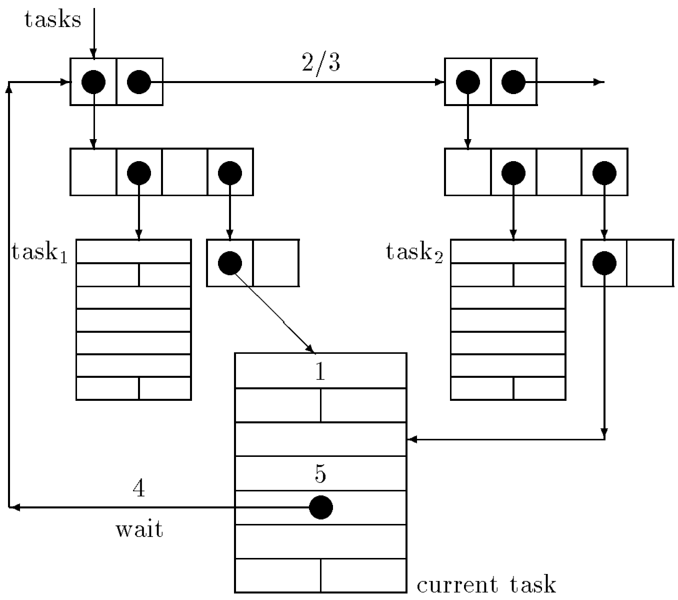

Figure 1: Waiting for a Result

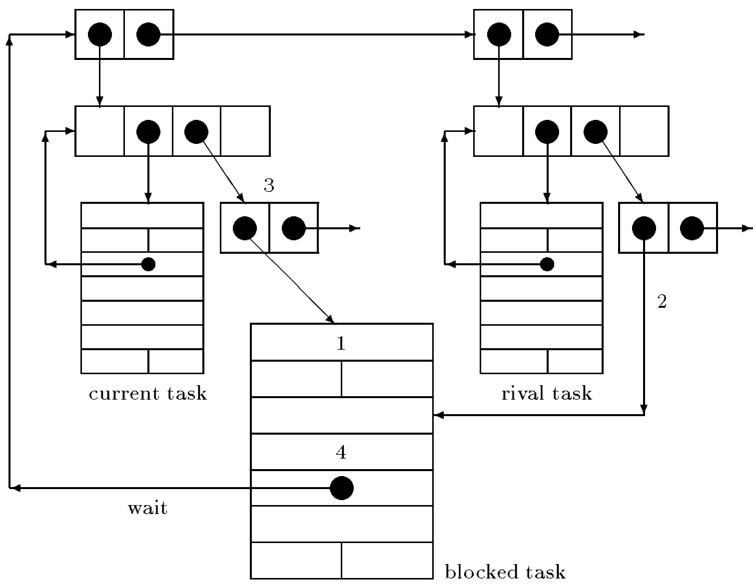

Figure 2: Delivering the Result

1. **Pick Task:** The semaphore associated to r is locked and the first task t waiting in the queue is determined. In order to prevent a deadlock situation, r is unlocked before t is locked and then locked again. Without this detour, a rival task delivering its result to t might already have locked t and itself try to lock r in order to remove t from the waiting queue of r.

2. **Check Task:** If t has disappeared from the waiting queue of r, the corresponding task has in the meantime received its result from a slightly faster rival task. Otherwise, the current task has "won" the race and may deliver its result to t.

3. **Remove Descriptor:** t is removed from the waiting queue of r and from the waiting queues of all other result descriptors that the task is waiting for (the `wait` field of t holds a list of these descriptors). This guarantees that only one result is delivered to t.

4. **Deliver Result:** The result value is written into the `value` field of t and r is written into the `wait` field of t. The corresponding task is then awaked and can extract the desired information from t.

After the result has been delivered to all waiting tasks, it is written into the `value` field of r. The `task` field of r is set to `AVAILABLE` and r is unlocked. The task then unlinks the task descriptor from the global task list, releases its workspace and terminates.

3.4 Implementation

Appendix A shows the fraction of the source code of the PACLIB kernel that implements the result delivery scheme described above:

- `pacShell` is the function that is executed by any PACLIB task. It runs the denoted user function with the corresponding arguments and then calls `pacResult`.
- `pacResult` delivers the result to all blocked tasks and finally stores the value into the result descriptor. The function uses several macros that perform the basic functions on result descriptors (e.g. `resLock`) and task descriptors (e.g. `taskLock`). The function `uWakeTask` awakes a blocked task.
- `pacWaitList` is the basic function on which `pacWait` is based on. All task handles (i.e. result descriptors) are linked into a single list `rlist`. The function `uBlockTask` blocks the execution of the current task.

Attention: The source code given in the appendix is a slightly beautified, condensed and restructured version of the actual source code. The actual implementation is a bit more fault-tolerant, more optimized and more general than the functions shown there. However, the explanation of these details goes beyond the scope of the paper and we consider them of minor importance for our goal. The code given in the appendix would basically run as well. However, please note that already at this level of discussion we have performed some simplifications.

3.5 Verification Goal

Our goal is to formally justify our confidence in the correctness of the implementation of the PACLIB result delivery scheme that has been presented in an informal way in this section and as a program in the appendix. We would like to verify that this non-deterministic scheme is correct for *any* program that applies `pacStart` and `pacWait` under *all* circumstances (i.e. possible interleavings of operations).

4 Temporal Logic and the Semantics of Programs

The formal framework we will use for reasoning is that of *temporal logic* as described in [Krö87]. The main reasons for this choice are the following:

- **Axiomatic Approach:** Temporal logic is an axiomatic approach to the formal treatment of parallel programs. It does not reason about the semantics of a program but about its properties. This is of particular importance since we want to deal with a whole class of (incompletely specified) programs.
- **Intuitive Reasoning:** Temporal logic allows *intuitive* reasoning about programs since it just models our usual knowledge about time relationships. Details that are "obvious" for a human being can be skipped if it would be tedious to derive them from the basic axioms.
- **Hoare's Calculus:** As a special case, we can incorporate Hoare's calculus for the verification of sequential programs into the temporal logic framework. This allows to reason about sequential program parts in the common way.

This section gives the essential core of [Krö87]. However, we describe the material in a language that is more amenable to computer scientists. Moreover, in Subsection 4.6 we introduce some concepts that make the reasoning process more practicable.

4.1 Temporal Logic

The main idea of temporal logic is to incorporate the notion of *time* into the framework of first-order predicate logic. We define time to be an infinite, discrete and linearly ordered sequence of time points with a smallest element:

$$\tau_0 < \tau_1 < \tau_2 < \tau_3 < \dots$$

The state of a universe $\mathcal{S}$ at each moment τ is defined by a valuation function η_τ that maps time-dependent quantities to values. Furthermore, a valuation function ξ maps time-invariant quantities to their denotations. We may thus define a *temporal structure* $\mathcal{K} = (\mathcal{S}, \xi, \eta)$ and a valuation function $\mathcal{K}_\tau$ that describes the semantics of a formula A at a given time point t. Please note that in general $\mathcal{K}_\tau(A) \neq \mathcal{K}_{\tau'}(A)$ for $\tau \neq \tau'$.

We define the semantics of atomic formulas and of the logical connectives as follows:

$$\begin{array}{lll}
\mathcal{K}_\tau(A) = \mathcal{S}^{(\xi,\eta_\tau)} & \multicolumn{2}{l}{\text{for every atomic formula } A} \\
\mathcal{K}_\tau(\neg A) = \mathrm{t} & \text{iff} & \mathcal{K}_\tau(A) = \mathrm{f} \\
\mathcal{K}_\tau(A \wedge B) = \mathrm{t} & \text{iff} & \mathcal{K}_\tau(A) = \mathrm{t} \text{ and } \mathcal{K}_\tau(b) = \mathrm{t} \\
\mathcal{K}_\tau(A \vee B) = \mathrm{t} & \text{iff} & \mathcal{K}_\tau(A) = \mathrm{t} \text{ or } \mathcal{K}_\tau(b) = \mathrm{t} \\
\mathcal{K}_\tau(A \rightarrow B) = \mathrm{t} & \text{iff} & \mathcal{K}_\tau(A) = \mathrm{f} \text{ or } \mathcal{K}_\tau(b) = \mathrm{t} \\
\mathcal{K}_\tau(A \leftrightarrow B) = \mathrm{t} & \text{iff} & \mathcal{K}_\tau(A) = \mathcal{K}_\tau(B)
\end{array}$$

For reasoning about time, we additionally introduce the unary *temporal operators* $\Box$ ("always") $\Diamond$ ("sometime") and $\bigcirc$ ("next time") and extend the valuation function $\mathcal{K}_\tau$ by

$$\begin{array}{lll}
\mathcal{K}_\tau(\Box A) = \mathrm{t} & \text{iff} & \mathcal{K}_{\tau'}(A) = \mathrm{t} \text{ for every } \tau' \geq \tau \\
\mathcal{K}_\tau(\Diamond A) = \mathrm{t} & \text{iff} & \mathcal{K}_{\tau'}(A) = \mathrm{t} \text{ for some } \tau' \geq \tau \\
\mathcal{K}_\tau(\bigcirc A) = \mathrm{t} & \text{iff} & \mathcal{K}_{\tau+1}(A) = \mathrm{t}
\end{array}$$

In other words, for a fixed moment τ, $\Box A$ is true, iff A will hold from τ on forever. The semantics of the other operators is then also immediately clear.

The quantifiers $\forall$ and $\exists$ are defined by

$$
\begin{array}{llll}
\mathcal{K}_\tau(\forall x(A)) = \mathrm{t} & \text{iff} & \mathcal{K}'_\tau(A) = \mathrm{t} \text{ for every } \mathcal{K}' = (S, \xi', \eta') \\
& & \text{with } \xi'(y) \stackrel{\mathcal{S}}{=} \xi(y) \text{ for every } y \text{ other than } x \\
\mathcal{K}_\tau(\exists x(A)) = \mathrm{t} & \text{iff} & \mathcal{K}'_\tau(A) = \mathrm{t} \text{ for some } \mathcal{K}' = (S, \xi', \eta') \\
& & \text{with } \xi'(y) \stackrel{\mathcal{S}}{=} \xi(y) \text{ for every } y \text{ other than } x
\end{array}
$$

We we also introduce the binary temporal operators `atnext`, `before` and `until` by

$$
\begin{array}{lll}
\mathcal{K}_\tau(A \texttt{ atnext } B) = \mathrm{t} & \text{iff} & \mathcal{K}_{\tau'}(A) = \mathrm{f} \text{ for every } \tau' \geq \tau \text{ or} \\
& & \mathcal{K}_\tau(A) = \mathrm{t} \text{ for the smallest } \tau' > \tau \text{ with } \mathcal{K}'_\tau(B) = \mathrm{t} \\
\mathcal{K}_\tau(A \texttt{ before } B) = \mathrm{t} & \text{iff} & \text{if } \mathcal{K}_{\tau'} = \mathrm{t} \text{ for some } \tau' > \tau \\
& & \text{then } \mathcal{K}_{\tau''} = \mathrm{t} \text{ for some } \tau'' \text{ with } \tau' > \tau'' > \tau \\
\mathcal{K}_\tau(A \texttt{ until } B) = \mathrm{t} & \text{iff} & \mathcal{K}_{\tau'} = \mathrm{t} \text{ for some } \tau' > \tau \text{ and} \\
& & \mathcal{K}_{\tau''} = \mathrm{t} \text{ for every } \tau'' \text{ with } \tau' > \tau'' > \tau
\end{array}
$$

Similar to classical first-order predicate logic, may introduce the notion of validity into temporal logic.

Definition 1 (Validity) *A formula A is called valid in a temporal structure $\mathcal{K}$ ($\mathcal{K}$-valid) if $\mathcal{K}_\tau(A) =$ t for every time τ. A is called valid if A is valid in every $\mathcal{K}$.*

Please note that the temporal notion of validity has one more "degree of freedom" than the classical notion: A is valid in the temporal sense only if it is true at every time point!

Similar to the transformation rules for logical connectives and quantifiers, there are many "temporal rules" like

$$
\begin{array}{lll}
\neg\Box A & \leftrightarrow & \Diamond\neg A \\
\bigcirc(A \rightarrow B) & \leftrightarrow & \bigcirc A \rightarrow \bigcirc B \\
\Box(A \rightarrow B) & \rightarrow & \Box A \rightarrow \Box B \\
\Box(A \wedge B) & \rightarrow & \Box A \wedge \Box B \\
\Diamond(A \vee B) & \rightarrow & \Diamond A \vee \Diamond B \\
\Box A \vee \Box B & \rightarrow & \Box(A \vee B) \\
\forall x(\bigcirc A) & \leftrightarrow & \bigcirc\forall x(A) \\
\exists x(\bigcirc A) & \leftrightarrow & \bigcirc\exists x(A) \\
\forall x(\Box A) & \leftrightarrow & \Box\forall x(A) \\
\exists x(\Diamond A) & \leftrightarrow & \Diamond\exists x(A) \\
A \texttt{ before } B & \leftrightarrow & (\neg B) \texttt{ atnext } (A \vee B)
\end{array}
$$

These rules are intuitively clear but can be easily derived from the definitions above e.g.

$$
\begin{array}{lll}
\mathcal{K}_\tau(\neg\Box A) = \mathrm{t} & \text{iff} & \text{not } \mathcal{K}_{\tau'}(A) = \mathrm{t} \text{ for every } \tau' \geq \tau \\
& \text{iff} & \mathcal{K}_u(A) = \mathrm{f} \text{ for some } \tau' \geq \tau \\
& \text{iff} & \mathcal{K}_u(\neg A) = \mathrm{t} \text{ for some } \tau' \geq \tau \\
& \text{iff} & \mathcal{K}_\tau(\Diamond\neg A) = \mathrm{t}
\end{array}
$$

For a much more comprehensive list of temporal rules, see [Krö87].

4.2 Parallel Programs

A parallel program Π consists of a set processes $\Pi_1 \ldots \Pi_p$ each of which is a sequential program composed from the basic "assignment" statement and the statement builders "sequence" (the ; operator), "conditional" (one and two branch if statement) and "iteration" (while loop). All statements operate on the same global set of "program variables". For the synchronization of parallel programs, there is a special statement

await C **then** $v := E$

where C is a boolean condition and E is an expression to be assigned to the variable v. The intuitive semantics of this construct is that $v := E$ can be executed only if C is true. Otherwise the process Π_i executing the **await** statement is blocked until C becomes true.

A program has the form

initial R;
cobegin $\Pi_1 \parallel \ldots \parallel \Pi_p$ **coend**

where R is a first-order formula called the *initialization condition* that describes the initial state of the program. As an example, take the following program

```
initial n = 0 ∧ sem
cobegin
(α0)   while i > 0 do
(α1)       await sem then sem := false
(α2)       stack[n] := i
(α2)       n := n + 1
(α3)       sem := true
(α4)       i := i − 1
       endwhile
(αe)   stop
||
(β0)   while j > 0 do
(β1)       await sem then sem := false
(β2)       stack[n] := j
(β2)       n := n + 1
(β3)       sem := true
(β4)       j := j − 1
       endwhile
(βe)   stop
coend
```

where two processes concurrently push items on a stack. A boolean variable *sem* (a "semaphore") is used for providing mutual exclusion of both processes during the stack access. Further on, we will use for semaphore operations the notation

P(*sem*) $\equiv$ **await** *sem* **then** *sem* := false
V(*sem*) $\equiv$ *sem* := true

which is more common in parallel programming than the **await** construct. The **stop** statement in the program above is actually a shortcut for

stop $\equiv$ **await** false

and always denotes the last statement in each parallel process beyond which execution cannot proceed. All statements in the program are labelled for a unique reference. The set of all labels in a program Π is denoted by $\mathcal{M}_\Pi$; the set of all labels without the trailing **stop** labels is denoted by $\overline{\mathcal{M}_\Pi}$.

4.3 Program Execution

The computational behavior of a parallel program is described by an *interleaving model* of execution where at each time point one atomic operation of a single process is executed.

Definition 2 (Atomic Operation) *An atomic operation is*

- *an assignment statement,*

- *an* **await** *statement (test of condition and execution of the assignment),*
- *a test of the condition in an* **if** *statement (and entry in the respective branch),*
- *a test of the condition in an* **while** *statement (and entry into or exit from loop).*

Definition 3 (Program State) *The state of a program Π at a time point τ is a tuple*

$$\eta_\tau = (\mu, \lambda_1, \ldots, \lambda_p, \kappa)$$

where

- *$\mu(a)$ is the value of the program variable a at time τ,*
- *λ_i is the label of the statement of Π_i that is ready to be executed at time τ,*
- *$\kappa \neq 0$ means that Π_κ is the process (and thus λ_κ is the statement) scheduled for execution at time τ,*
- *$\kappa = 0$ means that no Π_i may be scheduled at τ and therefore also not at any time $\tau' > \tau$ (deadlock or terminal state).*

The possible transitions from one program state to another will be described by a set of tuples (λ, C, λ') saying that, if the condition C is true, execution may proceed from statement λ to λ'.

Definition 4 (Transitions) *The set $trans(\Pi)$ of possible transitions of a sequential program Π is inductively defined on the structure of Π :*

1. **Stop:** If Π is the statement

   $\lambda_l :$ **stop**

   then $trans(\Pi) = \emptyset$.
2. **Assignment:** If Π is an assignment

   $\lambda_l : \ v := E$
   $\lambda_m : \ \ldots$

   then $trans(\Pi) = \{(\lambda_l, \mathrm{t}, \lambda_m)\}$.
3. **Wait Statement:** If Π is a wait statement

   $\lambda_l :$ **await** C **then** ...
   $\lambda_m : \ \ldots$

   then $trans(\Pi) = \{(\lambda_l, C, \lambda_m)\}$.
4. **Sequence:** If Π is a sequence

   $\lambda_l : \ \Pi_1$
   $\lambda_m : \ \Pi_2$
   $\lambda_n : \ \ldots$

   then $trans(\Pi) = trans(\Pi_1) \cup trans(\Pi_1)$.
5. **Two Branch Conditional:** If Π is a two branch conditional

   $\lambda_l :$ **if** C **then** $\lambda_m : \Pi_1$ **else** $\lambda_n : \Pi_2$ **endif**
   $\lambda_o : \ \ldots$

   then $trans(\Pi) = \{(\lambda_l, C, \lambda_m), (\lambda_l, \neg C, \lambda_m)\} \cup trans(\Pi_1) \cup trans(\Pi_2)$.
6. **One Branch Conditional:** If Π is a one branch conditional

   $\lambda_l :$ **if** C **then** $\lambda_m : \Pi_1$ **endif**
   $\lambda_n : \ \ldots$

   then $trans(\Pi) = \{(\lambda_l, C, \lambda_m), (\lambda_l, \neg C, \lambda_n)\} \cup trans(\Pi_1)$.

7. **While Loop** If Π is a while loop

$$\begin{array}{l}\lambda_l: \textbf{ while } C \textbf{ do } \lambda_m : \Pi_1 \textbf{ endwhile} \\ \lambda_n: \ \ldots\end{array}$$

then $trans(\Pi) = \{(\lambda_l, C, \lambda_m), (\lambda_l, \neg C, \lambda_n)\} \cup trans(\Pi_1)$.

In this informal definition, the trailing statement "..." denotes the successor of the current statement (such a successor exists and is unique for all statements but the final **stop** statement in a process). The exact definition of trans(Π) can be found in [Krö87].

Definition 5 (Statement Execution) *For every $\alpha \in \mathcal{M}_\Pi$, we define the execution condition*

$$E_\alpha = \begin{cases} C, & \text{if } \alpha: \texttt{await } C \texttt{ then } v := E \\ \text{true}, & \text{otherwise} \end{cases}$$

We may now define the structure of valid execution sequences of a program.

Definition 6 (Execution Sequence) *An execution sequence of Π is an infinite sequence $\eta = (\eta_0, \eta_1, \eta_2, \ldots)$ of program states such that*

1. **Initial State:** $\eta_0 = (\mu, \alpha_0^1, \ldots, \alpha_0^p, \kappa)$ *and* $\mathcal{K}_0(R) = \mathrm{t}$.
2. **Transition:** *If* $\eta_\tau = (\mu, \lambda_1 \ldots \lambda_\kappa \ldots \lambda_p, \kappa > 0)$ *then* $\eta_{t+1} = (\mu', \lambda_1 \ldots \lambda'_\kappa \ldots \lambda_p, \kappa')$ *and, for some formula C,* $(\lambda_\kappa, C, \lambda'_\kappa) \in trans(\Pi_\kappa)$ *and* $\mathcal{K}_\tau(C) = \mathrm{t}$.
3. **Terminal State:** *If* $\eta_\tau = (\mu, \lambda_1, \ldots, \lambda_p, 0)$ *then* $\eta_{\tau+1} = \eta_\tau$ *and* $\mathcal{K}_\tau(E_{\lambda_i}) = \mathrm{f}$ *for every* $1 \leq i \leq p$.

Please note that this definition does not give the semantics of the assignment and `await` statement since it does not describe the relationship between μ and μ' on a state transition.

4.4 Temporal Program Semantics

The language of temporal logic described in Subsection 4.1 is not yet sufficient to describe *program states* since it does not give any possibility to express information about the program counters $\lambda_1, \ldots, \lambda_p, \kappa$ in a program. For this purpose, we introduce the following set of propositional variables:

1. $\texttt{at}\{\lambda\}$ for every $\lambda \in \mathcal{M}_\Pi$.
2. λ for every $\lambda \in \overline{\mathcal{M}_\Pi}$.

The informal meaning of the proposition $\texttt{at}\{\lambda\}$ is "the statement λ is ready for execution in some process Π_i" while λ means "the statement λ is executed next". The information expressed by the program state

$$\eta_\tau = (\mu, \lambda_1, \ldots, \lambda_p, \kappa)$$

with $\kappa > 0$ can then be expressed by the formula

$$(v_1 = \mu(v_1)) \wedge \ldots \wedge (v_n = \mu(v_n)) \wedge \texttt{at}\{\lambda_1\} \wedge \ldots \wedge \texttt{at}\{\lambda_p\} \wedge \lambda_\kappa$$

where the v_i are the program variables occurring in Π.

In the notion of temporal logic, we may therefore interpret a program state η_τ as a valuation function

$$\begin{array}{lll} \eta_\tau(v) = \mu(v) & \text{for every program variable v} \\ \eta_\tau(\lambda) = \ \mathrm{t} & \text{iff} \quad \lambda = \lambda_\kappa \text{ and } \kappa \neq 0 \\ \eta_\tau(\texttt{at}\{\lambda\}) = \ \mathrm{t} & \text{iff} \quad \lambda = \lambda_j \text{ for some } j \end{array}$$

We may now apply the notion of validity defined in Subsection 4.1 to the special case of execution sequences.

Definition 7 (Π-Validity) *A formula A is Π-valid if it is true for every execution sequence η of Π (at any time τ).*

However, the restriction of temporal structures to execution sequences of programs can be also described by giving additional axioms and rules for deriving more formulas. We will distinguish between two kinds of axioms

1. Structural Axioms that are invariant for any program Π,
2. Computation Scheme Axioms that depend on Π.

It can be proved that a formula is Π-valid if it can be derived using these axioms [Krö87] where the following abbreviation is used:

Definition 8 (Initial and Terminal State) *For any process Π with $\mathcal{M}_\Pi = \{\alpha_0, \ldots, \alpha_l\}$ we define*

$$\begin{array}{lcl} \mathrm{start}_\Pi & \equiv & \mathtt{at}\{\alpha_0^i\} \wedge \ldots \wedge \mathtt{at}\{\alpha_0^p\} \wedge R \\ \mathrm{nil}_\Pi & \equiv & \neg\alpha_1 \wedge \neg\alpha_2 \wedge \ldots \neg\alpha_l \end{array}$$

The informal meaning of start_Π is "Execution of Π is in its initial state" while nil_Π says "No statement is executed in Π". The structural axioms are then the following:

Rule 1 (Basic Rule) *A is always true if it holds permanently beginning with the initial state of the execution sequence:*

$$\mathrm{start}_\Pi \rightarrow \Box A \vdash A.$$

Axiom 1 (Basic Axiom) *If no statement is executed, nothing changes:*

$$\mathrm{nil}_\Pi \wedge A \rightarrow \bigcirc(\mathrm{nil}_\Pi \wedge A).$$

Axiom 2 (Two Execute) *No two statements execute at the same time:*

$$\forall\lambda \neq \lambda'(\lambda \rightarrow \neg\lambda').$$

Axiom 3 (Execute) *An statement may only execute if it is ready to:*

$$\forall\lambda(\lambda \rightarrow \mathtt{at}\{\lambda\}).$$

Axiom 4 (Two Ready) *In every Π_k, no two statements are ready to execute at the same time:*

$$\forall k, l, m(\,\mathtt{at}\{\lambda_l^k\} \wedge l \neq m \rightarrow \neg\,\mathtt{at}\{\lambda_m^k\}).$$

Axiom 5 (Enabled) *If some statement is enabled, then some statement has to execute:*

$$\forall\lambda(\,\mathtt{at}\{\lambda\} \wedge E_\lambda \rightarrow \neg\mathrm{nil}_\Pi).$$

Axiom 6 (Ready) *An statement ready to execute but not executing remains ready:*

$$\forall\lambda(\,\mathtt{at}\{\lambda\} \wedge \neg\lambda \rightarrow \bigcirc\,\mathtt{at}\{\lambda\}).$$

Axiom 7 (End) *If Π_i has reached its end label then it remains there:*

$$\forall k(\,\mathtt{at}\{\lambda\}_e^k \rightarrow \bigcirc\,\mathtt{at}\{\lambda\}_e^k).$$

Axiom 8 (Change) *Every non-assignment and non-wait statement does not cause any variable changes:*

$$\forall\lambda \notin \{:=, \mathit{await}\}\ \forall P \notin \mathcal{L}_\mathrm{P}(\lambda \wedge P \rightarrow \bigcirc P).$$

The set of axioms describing the behavior of a particular program Π can be derived from the following axiom:

Axiom 9 (Computation Scheme) *For every $\lambda \in \mathcal{M}_\Pi$, the following holds*

$$\lambda \rightarrow (C_1 \wedge \bigcirc\, \mathtt{at}\{\lambda_1\}) \vee \ldots \vee (C_q \wedge \bigcirc\, \mathtt{at}\{\lambda_q\}).$$

where $(\lambda, C_1, \lambda_1), \ldots, (\lambda, C_q, \lambda_q)$ are all transitions with first statement λ in $\mathrm{trans}(\Pi)$.

Actually, this axiom is more general than necessary and can (according to the definition of $\mathrm{trans}(\Pi)$) be replaced by the following set of axioms that are more intuitive:

Axiom 10 (Assignment Statement) *If λ_l is an assignment statement i.e.*

$\lambda_l :\ v := E$
$\lambda_m :\ \ldots$

then execution proceeds from λ_l to λ_m:

$$\lambda_l \rightarrow \bigcirc\, \mathtt{at}\{\lambda_m\}.$$

Axiom 11 (Wait Statement) *If λ_l is a wait statement i.e.*

$\lambda_l :$ **await** C **then**...
$\lambda_m : \ldots$

then the condition C lets execution either suspend or proceed from λ_l to λ_m:

$$\lambda_l \rightarrow (C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_m\}) \wedge (\neg C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_l\}).$$

Axiom 12 (Two Branch Conditional) *If λ_l is the label of a two branch conditional i.e.*

$\lambda_l :$ **if** C **then** $\lambda_m :\ \ldots$ **else** $\lambda_n :\ \ldots$**endif**

then the condition C lets execution proceed from λ_l to one of the branches λ_m or λ_n, respectively:

$$\lambda_l \rightarrow (C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_m\}) \wedge (\neg C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_n\}).$$

Axiom 13 (One Branch Conditional) *If λ_l is the label of a one branch conditional i.e.*

$\lambda_l :$ **if** C **then** $\lambda_m :\ \ldots$ **endif**
$\lambda_n :\ \ldots$

then the condition C lets execution proceed from λ_l either to the branch λ_m or to the successor λ_n:

$$\lambda_l \rightarrow (C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_m\}) \wedge (\neg C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_n\}).$$

Axiom 14 (While Loop) *If λ_l is the label of a while loop i.e.*

$\lambda_l :$ **while** C **do** $\lambda_m : \ldots$ **endwhile**
$\lambda_n : \ldots$

then the condition C lets execution either proceed to the next iteration of the body λ_m or to the successor λ_n:

$$\lambda_l \rightarrow (C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_m\}) \wedge (\neg C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_l\}).$$

The effect of the assignment and the `await` statement is described by the following

Axiom 15 (Assignment) *For every statement*

$\lambda :\ v := E$

and every statement

$\lambda :$ `await` C `then` $v := E$

the following holds for every non-temporal formula P:

$$\lambda \wedge P_v(E) \rightarrow \bigcirc P.$$

The whole set of axioms describing the properties of a program Π is also called the *temporal semantics* of Π.

4.5 Description of Program Properties

There are basically two kinds of program properties that are of interest:

1. **Safety (or Invariance) Properties**,
2. **Liveness (or Eventuality) Properties**.

Invariance properties are expressed by formulas of the form

$$A \rightarrow \Box B.$$

For instance, the *partial correctness* of a program with precondition P and postcondition Q can be expressed by the formula

$$\mathrm{start}_\Pi \wedge P \rightarrow \Box((\,\mathtt{at}\{\alpha_e^1\} \wedge \ldots \wedge \mathtt{at}\{\alpha_e^p\}) \rightarrow Q)$$

saying that if Π starts with P then Q always holds in the terminal state.

Eventuality properties are expressed by formulas of the form

$$A \rightarrow \Diamond B.$$

As an example take the *termination* of a program with precondition P that can be written

$$\mathrm{start}_\Pi \wedge P \rightarrow \Diamond(\,\mathtt{at}\{\alpha_e^1\} \wedge \ldots \wedge \mathtt{at}\{\alpha_e^p\})$$

saying that if Π starts with P then it will eventually reach the terminal state.

The proof of invariant properties can be based on the following rule

Rule 2 (Invariant) *In order to prove "if A now holds then B always holds from now on", it suffices to prove that B follows from A and that B is invariant under every statement:*

$$(A \rightarrow B) \wedge \forall\lambda \in \overline{\mathcal{M}}_\Pi \; (\lambda \wedge B \rightarrow \bigcirc B) \;\vdash\; A \rightarrow \Box B.$$

For proving eventuality properties, we are faced with the problem that some "natural" program properties can not be derived from the set of axioms specified up to now. Consider e.g. the program

initial $a = 1$
cobegin
(α_0) $a := 0$
(α_e) **stop**
||
(β_0) **while** $a = 1$ **do**
(β_1) $b := 0$
endwhile
(β_e) **stop**
coend

Clearly we would claim that in any reasonable model of parallel execution Π should terminate sometime i.e. α_e should be reached. In other words,

$$\mathrm{start}_\Pi \rightarrow \Diamond\, \mathtt{at}\{\alpha_e\}$$

should be Π-valid but unfortunately it is not. The reason for this is that there is one execution sequence in which above formula is not true, namely that where forever β_0 and β_1 are alternately scheduled for execution. In order to overcome this deficiency, we introduce the notion of "fairness":

Definition 9 (Fairness) *An execution sequence $\eta = (\eta_0, \eta_1, \eta_2, \ldots)$ of a program Π is called fair it has the following property: If there are infinitely many states*

$$\eta_\tau = (\mu, \lambda_1, \ldots, \lambda_i, \ldots, \lambda_p, \kappa)$$

containing the same λ_i such that $\mathcal{K}_\tau(E_{\lambda_i}) = \mathrm{t}$, then $\kappa = i$ in infinitely many η_τ.

From now on, a formula A is considered Π-valid if it is valid in every *fair* execution sequence. Equivalently, this restriction can be described by the following

Axiom 16 (Fairness) *If an statement is enabled infinitely often then it will be executed sometime:*

$$\forall\lambda((\Box\Diamond(\,\mathtt{at}\{\lambda\} \wedge E_\lambda)) \rightarrow \Diamond\lambda).$$

This notion of fairness is also compatible with all process schedulers that apply a *round-robin* strategy with *preemptive execution* of processes (like the PACLIB scheduler).

Definition 10 (Process Execution) *For any process Π_i with $\overline{\mathcal{M}}_{\Pi_i} = \{\alpha_0^i, \ldots, \alpha_m^i\}$ we define the execution condition E^i as*

$$E^i \equiv (\,\mathtt{at}\{\alpha_0^i\} \wedge E_{\alpha_0^i}) \vee \ldots \vee (\,\mathtt{at}\{\alpha_0^e\} \wedge E_{\alpha_0^e})$$

The informal meaning of E^i is "one of the statements in Π^i is enabled for execution". This predicate is used in the following rule on which the proof of eventually properties is based.

Rule 3 (Well Founded Orderings) *To prove that B will eventually hold it suffices to prove that there is a well-founded ordering $(Z, \preceq)$ and a term z such that no statement in Π increases z with respect to this ordering, that there is for each value of z a ""helpful" process $\Pi_{h(z)}$ decreasing z and that $\Pi_{h(z)}$ will be eventually scheduled for execution:*

$$\begin{array}{ll} & \alpha \wedge A(z) \rightarrow \bigcirc(B \vee \exists z'(z' \preceq z \wedge A(z'))) \text{ for every } \alpha \in \overline{\mathcal{M}}_\Pi \backslash \overline{\mathcal{M}}_{\Pi_{h(z)}}, \\ & \alpha \wedge A(z) \rightarrow \bigcirc(B \vee \exists z'(z' \prec z \wedge A(z'))) \text{ for every } \alpha \in \overline{\mathcal{M}}_{\Pi_{h(z)}} \\ & \Box A(z) \rightarrow \Diamond(B \vee E^{\Pi_{h(z)}}) \\ \vdash & \exists z(A(z)) \rightarrow \Diamond B \end{array}$$

4.6 Time Constants

The standard framework of temporal logic does not give any possibility for explicitly referencing time points. [Krö87] says

> Consider now a proposition A. In order to describe the possible variety of the truth values of A at different times t the simplest linguistic means would be to introduce an explicit time parameter in the proposition and and denote it by $A(t)$. The main idea of the logical language we want to define, however, is to avoid just such an explicit occurrence of time. The reason is that we do not really want to be able to express assertions like "A is true at t_{17}". We rather want to have at hand logical *operators* which enable us to formulate new propositions about the truth values of A at time points which are related to some reference point (the *present time*) in particular ways.

We agree with the statement that we are not interested in expressing assertions like "A holds at t_{17}". However, we argue that time references are nevertheless useful for using

1. different "reference systems" in formulas,
2. a more general induction rule for proving.

The first item addresses the fact that even within a fixed temporal structure $\mathcal{K}$ the assertion

$$A$$

has not a unique meaning since it does not say *when* A holds. By just writing down A, it is not clear whether we want to say $\mathcal{K}_{17}(A) = \mathrm{t}$ or $\mathcal{K}_{23}(A) = \mathrm{t}$ or $\mathcal{K}_\tau(A) = \mathrm{t}$ where τ is the "present time". The usual solution is to consider A as a shortcut for

$$\Box A.$$

However, the definition of the $\Box$ operator

$$\mathcal{K}_\tau(\Box A) = \text{ t if } \mathcal{K}_{\tau'}(\Box A) = \text{t for every } \tau' > \tau$$

shows that we have shifted the problem only to another level. The denotation of $\Box A$ still depends on the reference point τ considered as the "present time". In program verification, the usual attitude is to consider the the start time of a program as the reference time 0 and to express all assertions relative to this time. Therefore, A is actually a shortcut for

$$\mathcal{K}_0(\Box A) = \text{t}.$$

If we want to reference another time point τ_A during a program run, we therefore have to find a unique characterization by a condition A on the state of the program variables and of the location counters such that

$$\mathcal{K}_{\tau_A}(A) = \text{t and } \mathcal{K}_\tau(A) = \text{ f for every } \tau \neq \tau_A.$$

The assertion "B holds at τ_A" is then expressed by $A \to B$. However, this indirect way of referencing time points makes propositions often difficult to understand. We have to remember that the formula is a shortcut for $\mathcal{K}_0(\Box(A \to B)) = \text{t}$ which can be reduced to

$$\mathcal{K}_{\tau_A}(B) = \text{t}.$$

what we actually would like to say. The problem is that the one-to-one correspondence between A and τ_A is not a priori clear from the formula. This is especially annoying if we want to switch to another reference time i.e. to formulate a set of formulas relatively to some τ_A. We therefore propose to directly reference particular time points by the introduction of time constants.

Definition 11 (Time Constants) *For every time τ there is a constant τ defined by*

$$\begin{aligned} \mathcal{K}_{\tau'}(\tau) \quad &= \text{t} \quad \text{if } \tau = \tau' \\ &= \text{f} \quad \text{if } \tau \neq \tau' \end{aligned}$$

A formula $\tau \to A$ is therefore a shortcut for $\mathcal{K}_0(\Box(\tau \to A))$ which can be reduced to

$$\begin{aligned} \mathcal{K}_0(\Box(\tau \to A)) = \text{ t } \quad &\text{if } \mathcal{K}_{\tau'}(\tau \to A) = \text{t for every } \tau' \geq 0 \\ &\text{if } \mathcal{K}_\tau(A) = \text{t} \end{aligned}$$

i.e. the interpretation of the formula is directly clear from its syntactic structure. In a similar way, the interpretation of $\tau_0 \to (A \texttt{ until } \tau_1)$ can be reduced to

$$\mathcal{K}_0(\Box(\tau_0 \to (A \texttt{ until } \tau_1))) = \text{ t if } \mathcal{K}_\tau(A) = \text{ t for every } \tau \text{ with } \tau_0 \leq \tau < \tau_1$$

which is immediately clear from the formula itself.

The purpose of time constants is therefore to constrain the reference time (respectively the time range) of a formula and to make this constraint explicit in the formula. We will therefore say that τ is the reference time of A in a context $\tau \to A$ and that the interval $[\tau_0 \dots \tau_1[$ is the time range of A in a context $\tau_0 \to (A \texttt{ until } \tau_1)$. We will in our verification only use such constrained formulas.

Furthermore, we introduce an additional structural program axiom that defines a special time constant for the start time of a program Π:

Axiom 17 (Start) *τ_Π is the start time of the program Π:*

$$\tau_\Pi \to \text{start}_\Pi.$$

This implies that in the context of program verification for each time τ the assertion $\tau > \tau_\Pi$ holds. By Axiom 8 (Initial State), we then know

$$\tau_\Pi \to \Box A \vdash A$$

The *definition* of a new time constant τ will always depend on previously defined time constants starting with τ_Π. This definition will be based on a characteristic condition A (as explained above) e.g. by saying

$$\tau_\Pi \to \tau \ \texttt{atnext}\ A$$

(where we have to prove that $\tau_\Pi \to \Diamond A$). After this definition, we may therefore use the symbol τ to reference the denoted time and apply the assertion $\tau \to A$.

A second advantage of introducing time constants is the possibility of applying a generalized induction rule. A proof guided by Rule 2 (Invariant) only allows conclusions from one statement to the next. We think that this view does not reflect the way most human beings intuitively reason about a program. Consider e.g. the program

initial $n = 0 \wedge N \geq 0 \wedge sem$
cobegin $\Pi_1 \parallel \ldots \parallel \Pi_p$ **coend**

where each Π_i is constructed as follows:

(α^i_0)	$\mathbf{P}(sem)$
(α^i_1)	**while** $n < N$ **do**
(α^i_2)	$\quad s_i := n$
(α^i_3)	$\quad n := n + 1$
(α^i_4)	$\quad \mathbf{V}(sem)$
(α^i_5)	$\quad a[s_i] := s_i$
(α^i_6)	$\quad \mathbf{P}(sem)$
	endwhile
(α^i_7)	$\mathbf{V}(sem)$
(α^i_e)	**stop**

Suppose that we want to verify

(1) $\quad \Box(\forall i(\texttt{at}\{\alpha^i_e\}) \to \forall j < N(a[j] = j))$.

i.e. that at the termination of the program $a[j] = j$ holds for every j.

Proof: For proving this assertion we describe the temporal behavior of the program by the formula

$$\begin{aligned}
A \equiv\ & \\
& (n \leq N, \\
& sem \to (B_n \wedge \forall i(C_i)), \\
& \neg sem \to \exists k(C_k \wedge \forall i \neq k(\neg C_k)), \\
& \forall i(\texttt{at}\{\alpha^i_1\} \to \neg sem \wedge B_n), \\
& \forall i(\texttt{at}\{\alpha^i_2\} \to \neg sem \wedge B_n \wedge n < N \wedge E), \\
& \forall i(\texttt{at}\{\alpha^i_3\} \to \neg sem \wedge B_n \wedge n < N \wedge s_i = n \wedge E), \\
& \forall i(\texttt{at}\{\alpha^i_4\} \to \neg sem \wedge B_{n-1} \wedge n \leq N \wedge s_i = n - 1 \wedge E), \\
& \forall i(\texttt{at}\{\alpha^i_5\} \to s_i < n), \\
& \forall i(\texttt{at}\{\alpha^i_7\} \to \neg sem \wedge n = N \wedge B_n), \\
& \forall i(\texttt{at}\{\alpha^i_e\} \to n = N \wedge B_n)).
\end{aligned}$$

where B_n, C_i and D are defined as

(2) $\quad B_n \equiv \forall j < n(a[j] \neq j \to \exists k(\texttt{at}\{\alpha^k_5\} \wedge s_k = j))$,

(3) $\quad C_i \equiv \texttt{at}\{\alpha^i_1, \ldots, \alpha^i_4, \alpha^i_7\}$,

(4) $\quad D \equiv \forall k(\neg\, \texttt{at}\{\alpha^k_e\})$.

The truth of (1) is a direct consequence of $\Box A$.

For proving $\Box A$, we have to prove by Rule 2 (Invariant) $\text{start}_\Pi \to A$ (which is easy) and, for each λ, $\lambda \wedge A \to \bigcirc A$. We will only show the case $\lambda = \alpha^i_4$ (for some i):

1. α_4^i and $n \leq N$ imply $\bigcirc n \leq N$.
2. α_4^i and A imply $\neg sem$, B_{n-1} and $s_i = n-1$. A and $\neg sem$ imply $\exists k(C_k \wedge \forall i \neq k(\neg C_k))$; α_4^i implies C_i and $\bigcirc\neg C_i$ and therefore $\bigcirc\forall i(\neg C_i)$. B_{n-1}, $s_i = n-1$ and $\bigcirc\,\mathtt{at}\{\alpha_5^i\}$ imply $\bigcirc B_n$.
3. α_4^i implies $\bigcirc sem$.
4. $\bigcirc\forall i(\neg C_i)$ and $\neg E$ imply $\bigcirc\neg\,\mathtt{at}\{\alpha_1^{i'}, \ldots, \alpha_4^{i'}, \alpha_7^{i'}, \alpha_e^{i'}\}$ for all i'. $\bigcirc\,\mathtt{at}\{\alpha_5^{i'}\}$ implies $\mathtt{at}\{\alpha_5^{i'}\}$, therefore $s_{i'} < n$ and thus $\bigcirc s_{i'} < n$. □

Please note that have only shown one of eight cases (for each α_l^i in the program)! A full verification of even this toy program becomes therefore very complex. This "statement-oriented" view is therefore not feasible for the verification of "real" parallel programs since the overview on the main lines of the proof easily goes lost. However, by the introduction of explicit time references we can use the following generalized induction rule that we think is more easier to use also for "semi-formal" proofs:

Rule 4 (Induction) *In order to prove "if A holds at τ_A then B always holds from τ_A on", it suffices to prove that B follows from A at τ_A and that B is true at every τ under the assumption that it has been true from τ_A until τ:*

$$(\tau_A \rightarrow (A \rightarrow B)) \wedge \forall\tau > \tau_A((\tau_A \rightarrow (B\ \mathtt{until}\ \tau)) \rightarrow (\tau \rightarrow B)) \ \vdash$$
$$\tau_A \rightarrow (A \rightarrow \Box B).$$

We will apply this rule to the program presented before. The main ideas there are

1. sem guarantees mutual exclusion on n,
2. B_n holds outside the critical region protected by sem,
3. $n = N$ holds at the end of the program.

Proof: The mutual exclusion property

(1) $\Box((sem \rightarrow \forall i(C_i)) \wedge (\neg sem \rightarrow \exists i(C_i \wedge \forall k \neq i(\neg C_k))))$

is easy to verify since α_0^i and α_6^i are the only entries and α_4^i and α_7^i the only exits from the critical region. Now let us prove that

(2) $\Box(sem \rightarrow B_n)$

By Rule 4 (Induction Rule), we have to prove $\tau_\Pi \rightarrow B_n$ which is clear since $\tau_\Pi \rightarrow n = 0$. Now take an arbitrary fixed time $\tau > \tau_\Pi$ and assume

(3) $\tau_\Pi \rightarrow ((sem \rightarrow B_n)\ \mathtt{until}\ \tau)$

(4) $\tau \rightarrow sem$

We have to prove $\tau \rightarrow B_n$. Let $\tau' = \tau - 1$. We only have to consider those cases where the truth value of $sem \rightarrow B_n$ may become false i.e. where $\tau' \rightarrow \neg sem$ (at the exits of critical regions) or where indeed $\tau' \rightarrow sem$ (and thus by the induction hypothesis $\tau' \rightarrow B_n$) but perhaps $\tau \rightarrow \neg B_n$. Actually, there are only the following cases are of interest:

1. $\boldsymbol{\tau' \rightarrow (\alpha_4^i \vee \alpha_7^i)}$: We know $\tau' \rightarrow \neg sem$. Let τ'' be the largest $\tau'' < \tau'$ with $\tau'' \rightarrow sem$. By the induction hypothesis, we then know $\tau'' \rightarrow B_n$. By the program structure, we know $\tau'' \rightarrow (\alpha_0^i \vee \alpha_6^i)$ and $\tau'' \rightarrow \bigcirc\alpha_1^i$. By (1), we know $\tau'' \rightarrow (s_i = n-1 \wedge B_{n-1})\ \mathtt{atnext}\ \alpha_4^i$ i.e. $\tau' \rightarrow (s_i = n-1 \wedge B_{n-1})$. We thus have $\tau \rightarrow B_n$.
2. $\boldsymbol{\tau' \rightarrow \alpha_5^i}$: If $\tau' \rightarrow sem$, we know by the induction hypothesis $\tau' \rightarrow B_n$. We also know $\tau' \rightarrow \mathtt{at}\{\alpha_5^i\}$ and $\tau' \rightarrow \bigcirc\neg\,\mathtt{at}\{\alpha_5^i\}$ but also $\tau' \rightarrow \bigcirc a[s_i] = s_i$. Thus $\tau' \rightarrow \bigcirc B_n$.

   If $\tau' \rightarrow \neg sem$, let τ'' be the largest $\tau'' < \tau'$ with $\tau'' \rightarrow sem$. By the induction hypothesis, we know $\tau'' \rightarrow B_n$. Since $\tau'' \rightarrow \neg sem\ \mathtt{until}\ \tau'$, it is by mutual exclusion and the program structure clear that, for any $\bar{n}$, $\tau'' \rightarrow n = \bar{n}$ implies $\tau' \rightarrow n \leq \bar{n} + 1$ and therefore $\tau' \rightarrow B_{n-1}$. This implies $\tau \rightarrow B_n$ with the above argument.

The fact that

(5) $\quad \tau_\Pi \rightarrow \Box \forall i(\,\mathtt{at}\{\alpha_e^i\} \rightarrow n = N)$

can be checked as follows. Let τ and i be arbitrary fixed such that

(6) $\quad \tau \rightarrow \mathtt{at}\{\alpha_e^i\}$.

By the program structure, it is clear that $\tau \rightarrow n \geq N$. Now assume $\tau \rightarrow n > N$. In this case, there exists some $\tau' < \tau$ and some i' such that

(7) $\quad \tau' \rightarrow (n = N \wedge \alpha_3^{i'})$.

By (1), this implies that for the largest $\tau'' < \tau'$ with $\tau'' \rightarrow \alpha_2^{i'}$ we have $\tau'' \rightarrow n = N$ and therefore $\tau'' \rightarrow \bigcirc\,\mathtt{at}\{\alpha_7^{i'}\}$ which contradicts (7). $\Box$

The application of Rule 4 (Induction) does of course principally not save any work; the complexity of a formal proof is as large as a proof carried through by Rule 2 (Invariant) (since in the induction step a case distinction over all statements is required). The main advantage of this new rule is that it allows to "jump back" in one step to a past time-point where something "interesting" has happened (e.g. to the entry of a critical region).

We may then apply the induction hypothesis to this time point and proceed to the "current" time point. The reasoning on the "uninteresting" intermediate points (e.g. within a critical region) may be then less formal. In the "statement-oriented" style of reasoning, such jumps are not possible. There the behavior of the program has to be described for each statement and verification has to proceed from one statement to the next.

For the use of Rule 4 (Induction), we introduce the "backward" oriented time operators $\triangle$ ("previous time") and `atprev`:

$$
\begin{array}{rcll}
\mathcal{K}_\tau(\triangle A) = \mathrm{t} & \text{iff} & \mathcal{K}_{\tau-1}(A) = \mathrm{t} \text{ and } \tau > 0 \\
\mathcal{K}_\tau(A \;\mathtt{atprev}\; B) = \mathrm{t} & \text{iff} & \mathcal{K}_{\tau'}(A) = \mathrm{t} \text{ for some } \tau' < \tau \\
 & \text{and} & \mathcal{K}_{\tau'}(B) = \mathrm{t} \text{ for the largest } \tau' < \tau \\
 & \text{with} & \mathcal{K}_{\tau'}(A) = \mathrm{t}
\end{array}
$$

Statements like "Let $\tau' = \tau - 1$; we know $\tau' \rightarrow A$" can now by replaced by $\tau \rightarrow \triangle A$.

For proving facts like $\tau_s \rightarrow (B\;\mathtt{until}\;\tau_e)$ we will apply

Rule 5 (Bounded Induction) *In order to prove "if A holds at τ_s then B always holds from τ_s up to τ_e", it suffices to prove that B follows from A at τ_s and that B is at every τ true before τ_e under the assumption that it has been true from τ_s on up to τ:*

$$
\begin{array}{l}
(\tau_s \rightarrow (A \rightarrow B)) \wedge \forall \tau_e > \tau > \tau_s((\tau_s \rightarrow (B\;\mathtt{until}\;\tau)) \rightarrow (\tau \rightarrow B)) \;\vdash \\
\quad \tau_s \rightarrow (A \rightarrow (B\;\mathtt{until}\;\tau_e)).
\end{array}
$$

This is a slight generalization of Rule 4 (Induction).

5 The Formal Model

5.1 The Overall Design

Our verification will be based on the following model program

$$
\begin{array}{lll}
\mathbf{\Pi} & \equiv & \mathbf{init}\; I \wedge J \\
 & & \mathbf{cobegin}\; \Pi_1 \;||\; \ldots \;||\; \Pi_p \;\mathbf{coend}
\end{array}
$$

where each process Π_k has the structure

$$\begin{array}{ll} & \dots \\ (\beta_1^k) & \mathbf{wait_k}(y_k, r_k, R_k) \\ (\beta_e^k) & \mathbf{nop} \\ & \dots \\ (\alpha_1^k) & \mathbf{result_k}(x_k) \\ (\alpha_e^k) & \mathbf{stop} \end{array}$$

Each process may contain an arbitrary number of *wait* calls (with arbitrary local computations between these calls) and is terminated by a *result* call. The informal meaning of these functions is

1. $\mathbf{wait(y, r, R)}$ waits for the result of one of the Π_i where i is in the set R. After returning from the call, r holds one of these identifiers and y holds the result of Π_r.
2. $\mathbf{result(x)}$ makes the process result x public to all processes that currently wait for this result or will at a later time point try to retrieve the result.

The **nop** (no operation) statement after the *wait* call only serves as a reference to the last instruction of *wait*, the **stop** statements after *result* terminates the program.

Please note that our model only describes the synchronization of PACLIB programs by `pacWait` but not the creation of new processes by `pacStart`. However, we will never use the process number p in our verification; the proof also works for an infinite number of processes and therefore also for programs that create new processes at runtime.

Furthermore, we do not demand that all the Π_k execute the same program. In fact, we do not at all make any assumptions about the structure of the processes before or after a particular *wait* call. Our verification will therefore be valid for *any* set of processes that use the *wait* construct.

Before we describe the initialization conditions I and J and the internal structure of **wait** and **result**, we will formulate the correctness theorem that we are going to verify.

5.2 The Correctness Theorem

The correctness of the program is established by the following theorem.

Theorem 1 (Correctness) *For every process j that calls $wait_j(y_j, r_j, R_j)$ the following holds:*

1. **No Result:** *If no process i in R_j has ever executed or will ever execute $result_i$, then j will not return from the $wait_j$ call.*
2. **Some Result:** *If some process i in R_j has executed or will execute $result_i$ then the following holds:*
   (a) **Wait Continuation:** *j returns from the $result_j$ call.*
   (b) **Result Delivery:** *after returning from $result_j$, the variable r_j holds some identifier i of R_j and the variable y_j holds the corresponding result x_i.*

Formally,

$$\begin{array}{l} \forall j\ \forall \tau_w \geq \tau_\Pi \forall \overline{R}\ \forall \overline{x}\ (\\ \quad (\tau_\Pi \rightarrow \Box \forall i(\alpha_1^i \rightarrow x_i = \overline{x}_i)) \wedge (\tau_w \rightarrow (\beta_1^j \wedge R_j = \overline{R}_j)) \ \rightarrow \\ \qquad (\tau_\Pi \rightarrow \forall i \in \overline{R}_j(\Box \neg \alpha_1^i)) \ \rightarrow \\ \qquad\quad (\tau_w \rightarrow \Box \neg \beta_e^j) \ \wedge \\ \qquad (\tau_\Pi \rightarrow \exists i \in \overline{R}_j(\Diamond \alpha_1^i)) \ \rightarrow \\ \qquad\quad (\tau_w \rightarrow \Diamond \beta_e^j) \ \wedge \\ \qquad\quad (\tau_w \rightarrow ((r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j}) \ \mathtt{atnext}\ \beta_e^j))). \end{array}$$

We will now explain this correctness theorem in more detail. The theorem contains the following mathematical constants and statement labels:

- j is the identifier of a process waiting for the result.
- τ_w is the time when j performs the $wait_j$ call.
- $\overline{R}_j$ is the "wait set" of j in this $wait_j$ call.
- $\overline{x}_i$ is the result of Π_i (for all i).
- β_1^j is the label of the $wait_j$ call.
- β_e^j is the label of the last statement of $wait_j$.
- α_1^i is the label of the $result_i$ call of Π_i (for all i).

The theorem formulates an assertion for each process j that performs a $wait_j$ call at some time point τ_w with some wait set $\overline{R}_j$.

The first assumption

$$\tau_\Pi \rightarrow \Box\forall i(\alpha_1^i \rightarrow x_i = \overline{x}_i)$$

declares the result of each process Π_i as the value of the program variable x_i at the time at which Π_i calls $result_i$. The result is associated to the mathematical constant $\overline{x}_i$, which is necessary because the value of x_i needs not be the same any more at the time when the $wait_j$ call returns the result of Π_i.

The second assumption

$$\tau_w \rightarrow (\beta_1^j \wedge R_j = \overline{R}_j)$$

defines τ_w to be the time at which Π_j calls $wait_j$ and associates the contents of the program variable R_j to the mathematical constant $\overline{R}_j$. This is required since the value of R_j may be changed during the program run.

The theorem then describes two cases:

1. $\boldsymbol{\tau_\Pi \rightarrow \forall i \in \overline{R}_j(\Box\neg\alpha_1^i)}$

   is the case when no process $i \in \overline{R}_j$ has ever called $result_i$ (i.e. *before* $wait_j$ is called) or will ever call $result_i$ (i.e. *after* $wait_j$ has been called). The reference time of the assumption is therefore the program start time τ_Π and the $\Box$ operator refers to the total runtime of the program. The conclusion

   $$\tau_w \rightarrow \Box\neg\beta_e^j$$

   says that Π_j will never return from this particular $wait_j$ call (but it does not say anything about former $wait_j$ calls of Π_j). The reference time of the conclusion is therefore the time τ_w when $wait_j$ is called and the $\Box$ operator only refers to the future of τ_w (but not to the time between τ_Π and τ_w).

2. $\boldsymbol{\tau_\Pi \rightarrow \exists i \in \overline{R}_j(\Diamond\alpha_1^i)}$

   is the case when some process $i \in \overline{R}_j$ has terminated before τ_w or will terminate after τ_w. The reference time of the assumption is therefore τ_Π and the $\Diamond$ operator refers to the total runtime of the program. The first conclusion

   $$\tau_w \rightarrow \Diamond\beta_e^j$$

   says that Π_j will return from this particular $wait_j$ call, i.e. it will execute β_e^j *after* τ_w. The second conclusion

   $$\tau_w \rightarrow ((r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j}) \texttt{ atnext } \beta_e^j)$$

says that after returning from this $wait_j$ call r_j holds one of the process identifiers of $\overline{R_j}$ and that y_j equals the result of this process. The reference time of this statement is therefore τ_w. It is not necessary to express by extra assertion that Π_{r_j} has terminated, since the only information about $\overline{x}_i$ can be taken from the assumption that associates $\overline{x}_i$ to the value of x_i at the termination time of Π_i.

Please note that the index i is always used if we refer to the role of a process Π_i as a deliverer of a result in $result_i$ while the index j is used if we refer to the role of some Π_j as a receiver of a result in $wait_j$. The details of $result_i$ and $wait_j$ are described in the next subsection.

5.3 The Detailed Structure

The following global variables are used in the model program for the communication and synchronization between processes:

- $value_i$ holds the result of Π_i after Π_i has terminated. This variable models the `value` field of a Paclib result descriptor.
- $avail_i$ is a boolean flag signalling that $value_i$ holds the result of Π_i. This variable models the `avail` field of a Paclib result descriptor.
- W_i is a set that holds the identifiers of all processes being blocked on an attempt to receive the result of Π_i. This variable models the queue defined by the `wait/value` fields of a Paclib result descriptor.
- $wsem_i$ is a binary semaphore that provides mutual exclusion during the access to W_i. This variable models the `sem` field of a Paclib result descriptor.
- R_j is the set that holds all the processes that some Π_j waits for. This variable one the one hand is the input argument of the $wait_j$ call, on the other hand it models the `wait` field of a Paclib task descriptor.
- $rsem_j$ is a binary semaphore that provides synchronization between all Π_i that may deliver their result to Π_j. This variable models the `sem` field of a Paclib task descriptor.
- run_j is a binary semaphore used to block a process Π_j when none of the Π_i with $i \in \overline{R_j}$ has yet terminated. This variable therefore models the dequeuing of a task descriptor from the scheduling queue.

This model is a simplification of the actual data representation in the Paclib kernel, in particular concerning the following features:

1. **Data Structures:** The task queue in a Paclib result descriptor and the task list in a Paclib task descriptor are modelled by the mathematical sets W_i and R_j, respectively. Hence, modifications of queues and lists are modelled by set operations. This may essentially change the semantics of the program as discussed below.
2. **Result Descriptor:** The `value` field in a Paclib result descriptor serves as the tail of the `wait` queue before it actually receives the result of the process. In the model program, this role is played by two distinct variables $value_i$ and W_i. This also has some crucial consequences on the program model as discussed below.
3. **Task Descriptor:** When a Paclib task is blocked, the `wait` field of the task descriptor contains the set of tasks that the blocked task waits for. After it has been awaked, the `wait` field holds the identifier of the delivering task and the `value` field holds the result. Since in the model program all variables are globally known to all processes, Π_i directly reads R_j to determine the processes that Π_j is waiting for and writes itself i and x_i into r_j and y_j, respectively. We will ignore this issue in our verification.

4. **Life Time:** In the model program, all variables have an infinite life time and may be accessed even after some processes have terminated. In the PACLIB kernel, the local variables of a process and the contents of a task descriptor may be not accessed any more after the process has terminated since the process workspace has been freed and the contents of the corresponding memory cells are undefined. Since this problem is closely related to many low level details of process creation and termination, we will not deal with it in our verification.

Before we discuss the model in more detail, let us take a look at the internal structure of $result_i$ and $wait_j$ in Figure 3 and Figure 4, respectively.

- $result_i$ first writes the result x_i into the variable $value_i$ and sets the $avail_i$ flag. It then temporarily locks $wsem_i$ to receive one process $j \in W_i$ blocked on an attempt to receive the result of Π_i. Afterwards it locks $rsem_j$ and determines whether j is still in W_i or if another process $\Pi_{i'}$ has already delivered its result to Π_j and has removed j from W_i. If j is still in W_i, Π_i has "won the race" and iteratively removes j from the wait sets $W_{i'}$ of all $\Pi_{i'}$ that Π_j was waiting for. Afterwards, Π_i delivers its result to Π_j, awakes the task by calling $\mathbf{V}(run_j)$ and continues the delivery of results until all blocked task have been awaked.

- $wait_j$ first locks $rsem_j$ to prevent any process Π_i from delivering its result to Π_j while Pi_j is not yet blocked. Then Π_j iteratively examines the state of each $avail_i$ with $i \in R_j$. If the $avail_i$ flag is not yet set, Π_j adds j to the corresponding wait set W_i and continues the search. If none of the $avail_i$ flags is set, Π_j unlocks $rsem_j$ and blocks itself by calling $\mathbf{P}(run_j)$. After it has been awaked by some Π_i, the results are available by $r_j = i$ and $y_j = x_i$. However, if some of the Π_i has already delivered its result, Π_j itself sets r_j and x_j and removes j from all W_i to which it has added j before.

The algorithm itself reflects more or less the PACLIB program given in Appendix A with one major exception: while `pacWait` fills the `value` and the `avail` field of its result descriptor at the end of the program, the function $wait_j$ defines $value_j$ and $avail_j$ at the beginning. The reason for this is the following:

1. `pacWait` uses the `value` field to store the tail pointer of the `wait` queue before the result is available. Since other tasks may remove items from this queue as long as this queue is not empty, the result may be written into the `value` field only at the end of the function (when all blocked tasks have been processed).

2. It can *not* be proved that a $wait_j$ call always returns on the basis of a pure set interpretation of W_j if the $value_i$ variable is filled at the end of the program. We either have to add an axiom that asserts that W_j fulfills the "first in, first out" property of queues or restructure the program in the presented way.

We will discuss this issue in Section 6.3 in more detail. The last point that we have to mention is the definition of the operations **add** and **rem** that are used for operations on the sets W_j. In the PACLIB kernel, the `wait` queues in the PACLIB result descriptors are the only compound data structures that are modified by different tasks. The purpose of the semaphores in these descriptors is to guarantee the consistent modification of these queues.

However, in our program we have modelled the `wait` queues as sets W_j to which the set operations $\cup$ and $\setminus$ may be applied. However, these set operations are atomic and need no protection at all! Even, if we keep the protecting semaphores in the program and prove that they do not cause any harm (like deadlocks), we may and will apply the fact that sets can be atomically updated by one statement.

In order to make the model more realistic, we therefore replace the set operations $\cup$ and $\setminus$ on W_j by the functions **add** and **rem** defined by the following code sequences:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{result_i}(x_i) \equiv} \\
 & \\
(\alpha^i_1) & value_i := x_i \\
(\alpha^i_2) & avail_i := \text{true} \\
 & \\
(\alpha^i_3) & \mathbf{P}(wsem_i) \\
(\alpha^i_4) & \mathbf{while}\ W_i \neq \emptyset\ \mathbf{do} \\
(\alpha^i_5) & \quad w_i \in W_i \\
(\alpha^i_6) & \quad \mathbf{V}(wsem_i) \\
 & \\
(\alpha^i_7) & \quad \mathbf{P}(rsem_{w_i}) \\
 & \\
(\alpha^i_8) & \quad \mathbf{P}(wsem_i) \\
(\alpha^i_9) & \quad deliver_i := w_i \in W_i \\
(\alpha^i_{10}) & \quad \mathbf{V}(wsem_i) \\
 & \\
(\alpha^i_{11}) & \quad \mathbf{if}\ deliver_i\ \mathbf{then} \\
 & \\
(\alpha^i_{12}) & \qquad S_i := R_{w_i} \\
(\alpha^i_{13}) & \qquad \mathbf{while}\ S_i \neq \emptyset\ \mathbf{do} \\
(\alpha^i_{14}) & \qquad\quad s_i \in S_i \\
(\alpha^i_{15}) & \qquad\quad \mathbf{P}(wsem_{s_i}) \\
(\alpha^i_{16}) & \qquad\quad \mathbf{rem}(W_{s_i}, w_i) \\
(\alpha^i_{17}) & \qquad\quad \mathbf{V}(wsem_{s_i}) \\
(\alpha^i_{18}) & \qquad\quad S_i := S_i \backslash \{s_i\} \\
 & \qquad \mathbf{endwhile} \\
 & \\
(\alpha^i_{19}) & \qquad y_{w_i} := x_i \\
(\alpha^i_{20}) & \qquad r_{w_i} := i \\
(\alpha^i_{21}) & \qquad \mathbf{V}(run_{w_i}) \\
 & \\
 & \quad \mathbf{endif} \\
 & \\
(\alpha^i_{22}) & \quad \mathbf{V}(rsem_{w_i}) \\
 & \\
(\alpha^i_{23}) & \quad \mathbf{P}(wsem_i) \\
 & \mathbf{endwhile} \\
(\alpha^i_{24}) & \mathbf{V}(wsem_i) \\
 & \\
(\alpha^i_e) & \mathbf{stop}
\end{array}
$$

Figure 3: Result Process

$\mathbf{wait_j}(y_j, r_j, R_j) \equiv$

(β^j_1) $P_j := R_j$

(β^j_2) $\mathbf{P}(rsem_j)$
(β^j_3) **while** $P_j \neq \emptyset \wedge \neg found_j$ **do**

(β^j_4) $p_j \in P_j$
(β^j_5) $\mathbf{P}(wsem_{p_j})$

(β^j_6) **if** $\neg avail_{p_j}$ **then**

(β^j_7) $\mathbf{add}(W_{p_j}, j)$
(β^j_8) $\mathbf{V}(wsem_{p_j})$
(β^j_9) $P_j := P_j \backslash \{p_j\}$

else
(β^j_{10}) $\mathbf{V}(wsem_{p_j})$

(β^j_{11}) $y_j := value_{p_j}$
(β^j_{12}) $r_j := p_j$
(β^j_{13}) $found_j :=$ true

(β^j_{14}) $Q_j := R_j \backslash P_j$
(β^j_{15}) **while** $Q_j \neq \emptyset$ **do**
(β^j_{16}) $q_j \in Q_j$
(β^j_{17}) $\mathbf{P}(wsem_{q_j})$
(β^j_{18}) $\mathbf{rem}(W_{q_j}, j)$
(β^j_{19}) $\mathbf{V}(wsem_{q_j})$
(β^j_{20}) $Q_j := Q_j \backslash \{q_j\}$
endwhile
endif

endwhile
(β^j_{21}) $\mathbf{V}(rsem_j)$

(β^j_{22}) **if** $found_j$ **then**
(β^j_{23}) $found_j :=$ false
else
(β^j_{24}) $\mathbf{P}(run_j)$
endif

(β^j_e) **nop**

Figure 4: Wait Process

(λ_l^p) $\quad$ $\mathbf{add}(W, e) \equiv$

$(\lambda_{l_a}^p)$ $\quad$ $T_p := W$
$(\lambda_{l_b}^p)$ $\quad$ $W := \ldots$
$(\lambda_{l_c}^p)$ $\quad$ $W := T_p \cup \{e\}$

(λ_l^p) $\quad$ $\mathbf{del}(W, e) \equiv$

$(\lambda_{l_a}^p)$ $\quad$ $T_p := W$
$(\lambda_{l_b}^p)$ $\quad$ $W := \ldots$
$(\lambda_{l_c}^p)$ $\quad$ $W := T_p \backslash \{e\}$

where T_p is a new auxiliary variable. During the execution of these code sequences, there exists one *inconsistent* state during which we have no information on the value of W. If we are now able to verify the correctness of the model program, we may assume that the inconsistent state during the modifications of the `wait` queues does not cause any harm. For all other sets, that are either only used by one process or not modified at all, we will use the mathematical set operations.

Finally, let us now define the initialization condition $I \wedge J$ of Π where I defines the initial states of all "*result*-related" program variables and J defines the initial states of all "*wait*-related" variables:

$$I \equiv \forall i(wsem_i \wedge \neg avail_i \wedge W_i = \emptyset)$$
$$J \equiv \forall j(rsem_j \wedge \neg found_j \wedge \neg run_j)$$

We assume that these variables will be only used within the functions $result_i$ and $wait_j$ and are in particular not modified elsewhere in Π. The main lines of the proof and the central theorems are presented in the next section.

6 The Verification

In this section, we will present the main lines of the verification of Theorem 1 (Correctness). The proof is based on four central theorems that can be informally described as follows:

1. **Theorem 2 (No Result)** says that the program flags $found_j$ and run_j will be never set to true, if no process Π with $i \in R_j$ terminates.
2. **Theorem 3 (Found)** says that if Π_j has set the found flag, it has itself found the result of some terminated process Π_i.
3. **Theorem 4 (Awake)** says that, if some Π_i has set run_j to true, Π_i has delivered its result to Π_j.
4. **Theorem 5 (Sometime)** says that some Π_i will eventually set run_j to true, if Π_j is blocked.

Please note that Theorem 2, Theorem 3 and Theorem 4 describe *invariance properties* while Theorem 5 describes an *eventuality property* of Π .

Theorem 2 is the basis of Proposition 1 (No Result) of the correctness theorem; Theorem 3 and Theorem 4 together establish the correctness of Proposition 2b (Result Delivery); last but not least Theorem 5 is the main constituent of Proposition 2a (Wait Continuation).

In the whole proof, in all theorems and lemmas, we assume that j, $\tau_w \geq \tau_\Pi$, $\overline{R}$ and $\overline{x}$ are arbitrary fixed such that

(1) $\quad$ $\tau_s \rightarrow \Box \forall i(\alpha_1^i \rightarrow x_i = \overline{x}_i)$,
(2) $\quad$ $\tau_w \rightarrow (\beta_1^j \wedge R_j = \overline{R}_j)$.

Furthermore, we will use the predicate

$$\texttt{elem}_i(j, W) \equiv (wsem_i \to j \in W) \wedge (\neg wsem_i \to (j \in W \texttt{ atnext } wsem_i)).$$

which informally says that, if no process currently updates the set W, j is in W. However, if W is currently modified, then i will be in W when W is consistent again. In general, we use this predicate instead of $\in$ to describe properties of W_i because an assertion $j \in W_i$ that has held before W_i has been modified and will hold after the modification has been completed does not hold during the modification!

In the following three subsections, we will verify the three conclusions of the correctness theorem based on the theorems presented above. We will prove these theorems based on a set of lemmas described and proved in Appendix B. The reader should understand the contents of these lemmas (not their proofs) before reading the following subsections. The proofs of these lemmas actually represent the technically difficult parts of the verification.

6.1 No Result

We will prove Proposition 1 of Theorem 1 (Correctness). We assume that no process in $\overline{R}_j$ delivers a result

(3) $\tau_\Pi \to \forall i \in \overline{R}_j(\Box\neg\alpha_1^i)$

and want to prove that process j will not return from the $wait_j$ call

(4) $\tau_w \to \Box\neg\beta_e^j$.

By $\tau_\Pi \leq \tau_w$ and Axiom 3 (Execute), it suffices to prove

(5) $\tau_\Pi \to \Box\neg\,\texttt{at}\{\beta_e^j\}$.

By Rule 4 (Induction), we first have to show

(6) $\tau_\Pi \to \neg\,\texttt{at}\{\beta_e^j\}$

which is a direct consequence of $start_\Pi$ and Axiom 4 (Two Ready). Now let time $\tau > \tau_\Pi$ be arbitrary fixed. Assuming

(7) $\tau_\Pi \to (\neg\,\texttt{at}\{\beta_e^j\}\texttt{ until }\tau)$

we have to prove

(8) $\tau \to \neg\,\texttt{at}\{\beta_e^j\}$.

By the program structure and (7), we may assume

(9) $\tau \to \triangle(\beta_{23}^j \vee \beta_{24}^j)$.

since otherwise (8) is immediately fulfilled. We will now distinguish two cases:

Case $\tau \to \triangle\beta_{23}^j$

By Theorem 2 (No Result), we have

(10) $\tau \to \triangle\neg found_j$.

This, by Lemma 16 (Reset), contradicts the case assumption.

Case $\tau \to \triangle\beta_{24}^j$

By Theorem 2 (No Result), we have

(11) $\tau \to \triangle\neg run_j$.

This, by Axiom 11 (Wait Statement), contradicts the case assumption. □

We will now prove the central theorem of this subsection which says that, if no Π_i with $i \in R_j$ terminates, the flag $found_j$ and the binary semaphore run_j will be never set.

Theorem 2 (No Result) *If no process in $\overline{R}_j$ terminates, j will not find a result and be blocked forever i.e.*

$$(\tau_\Pi \rightarrow \Box \forall i \in \overline{R}_j(\neg\alpha_1^i)) \rightarrow (\tau_\Pi \rightarrow \Box(\neg found_j \wedge \neg run_j)).$$

Proof: Assume

(12) $\tau_\Pi \rightarrow \Box \forall i \in \overline{R}_j(\neg\alpha_1^i)$.

By Rule 4 (Induction), we first have to prove

(13) $\tau_\Pi \rightarrow (\neg found_j \wedge \neg run_j)$

which is a consequence of $start_\Pi$. Now let time $\tau > \tau_\Pi$ be arbitrary fixed. Assuming

(14) $\tau_\Pi \rightarrow ((\neg found_j \wedge \neg run_j)\ \texttt{until}\ \tau)$

we have to prove

(15) $\tau \rightarrow (\neg found_j \wedge \neg run_j)$.

By the structure of the program, we may assume

(16) $\tau \rightarrow \triangle(\beta_{13}^j \vee \exists i(\alpha_{21}^i \wedge w_i = j))$.

since these are the only statements modifying $found_j$ and run_j. However, $\tau \rightarrow \triangle\beta_{13}^j$ contradicts Lemma 10 (If Available) and Lemma 11 (Not Available). Furthermore, $\tau \rightarrow \triangle\exists i(\alpha_{21}^i \wedge w_i = j)$ contradicts by the program structure (12). $\Box$

6.2 Result Delivery

We will prove Proposition 2b of Theorem 1 (Correctness). We assume

(17) $\tau_\Pi \rightarrow \exists i \in \overline{R}_j(\Diamond\alpha_1^i)$.

and want to prove

(18) $\tau_w \rightarrow ((r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})\ \texttt{atnext}\ \beta_e^j)$.

In case of

(19) $\tau_w \rightarrow \Box\neg\beta_e^j$

we are done. Otherwise, by the program structure there exist times τ_d, τ_e, τ_f, τ_r such that

(20) $\tau_w \rightarrow (\tau_d\ \texttt{atnext}\ \beta_2^j)$,

(21) $\tau_w \rightarrow (\tau_e\ \texttt{atnext}\ \beta_{21}^j)$,

(22) $\tau_w \rightarrow (\tau_f\ \texttt{atnext}\ \texttt{at}\{\beta_e^j\})$,

(23) $\tau_w \rightarrow (\tau_r\ \texttt{atnext}\ \beta_e^j)$.

These definitions will be used in this section and all associated lemmas. We now can reformulate (18) as

(24) $\tau_r \rightarrow (r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})$.

By Axiom 3 (Execute), we know

(25) $\tau_r \rightarrow \texttt{at}\{\beta_e^j\}$.

We will prove (24) by induction on the number t of times that β_e^r was not scheduled for execution i.e. on $t = \tau_r - \tau_f$. The base case of the induction $\tau_r = \tau_f$ characterized by

(26) $\tau_r \rightarrow \triangle(\beta_{23}^j \vee \beta_{24}^j)$

will be shown in the following two subsections. For the induction step, let us assume

(27) $\tau_r \rightarrow \triangle(\ \texttt{at}\{\beta_e^j\} \wedge r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})$.

By Lemma 30 (Exit), we also know

(28) $\tau \to \triangle\forall i(\,\mathtt{at}\{\alpha_6^i \ldots \alpha_{21}^i\} \to w_i \neq j)$

i.e. r_j and y_j cannot be modified by any Π_i. Therefore we have (24).

We will further on deal with the two cases of the induction base.

6.2.1 Case $\tau_r \to \triangle\beta_{23}^j$

By Lemma 16 (Reset), we know

(29) $\tau_r \to \triangle found_j$.

By Theorem 3 (Found), we then know

(30) $\tau_r \to \triangle(r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})$.

By the program structure, we then have (24).

We will now prove the first central theorem of this subsection which says that if Π_j has set the $found_j$ flag it has found the result of some Π with $i \in R_j$.

Theorem 3 (Found) *If j has set the found flag, it has indeed found a result:*

$$\tau_w \to ((found_j \to (r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j}))\ \mathtt{until}\ \tau_r).$$

Proof: Define

(31) $E \equiv found_j \to (r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})$.

By Rule 5 (Bounded Induction), we first have to show

(32) $\tau_w \to E$

which is a consequence of Lemma 19 (Entry). Now let τ be arbitrary fixed with $\tau_w < \tau < \tau_r$ and assume

(33) $\tau_w \to (E\ \mathtt{until}\ \tau)$.

We have to prove

(34) $\tau \to E$.

We may assume

(35) $\tau \to \triangle(\beta_{11}^j \vee \beta_{12}^j \vee \beta_{13}^j \vee \exists i(w_i = j \wedge (\alpha_{19}^j \vee \alpha_{20}^j)))$

since these are the only statements modifying y_j, r_j or $found_j$.

Case $\tau \to \triangle(\beta_{11}^j \vee \beta_{12}^j)$

By Lemma 15 (Not Found), we have

(36) $\tau \to \triangle\neg found_j$

and hence, by the program structure,

(37) $\tau \to \neg found_j$.

Case $\tau \to \triangle\beta_{13}^j$

By Lemma 10 (If Available), we know

(38) $\tau \to \triangle avail_{p_j}$

and by Lemma 5 (Result Exclusion)

(39) $\tau \rightarrow \triangle(r_j = p_j)$.

This implies by Lemma 12 (Process Set) and Lemma 13 (Result)

(40) $\tau \rightarrow \triangle(r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})$.

Then we are by the program structure done.

Case $\tau \rightarrow \triangle(w_i = j \wedge (\alpha^i_{19} \vee \alpha^i_{20}))$

By Lemma 22 (Lock), we know

(41) $\tau \rightarrow (\texttt{elem}_i(j, W_i)\ \texttt{atprev}\ \alpha^i_7)$.

Furthermore, by Axiom 11 (Wait Statement) we know

(42) $\tau \rightarrow (rsem_j\ \texttt{atprev}\ \alpha^i_7)$.

This implies by Lemma 28 (Result Loop)

(43) $\tau \rightarrow ((\neg found_j \wedge \neg run_j \wedge \texttt{at}\{\beta^j_{22}, \beta^j_{24}\})\ \texttt{atprev}\ \alpha^i_7))$.

By Lemma 5 (Result Exclusion) and the program structure we then know

(44) $\tau \rightarrow \neg found_j$. □

6.2.2 Case $\tau \rightarrow \triangle\beta^j_{24}$

By Axiom 11 (Wait Statement), we know

(45) $\tau \rightarrow \triangle run_j$.

Thus we know by Theorem 4 (Awake)

(46) $\tau \rightarrow \triangle(r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})$

By the program structure, we then have (24). □

We will now prove the second central theorem of this subsection which says that if some Π_i sets run_j it has delivered its result to Π_j.

Theorem 4 (Awake) *If process j is waked up by some process i, then j has received the result x_i:*

$$\tau_w \rightarrow ((run_j \rightarrow (r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j}))\ \texttt{until}\ \tau_r).$$

Proof: Define

(47) $F \equiv run_j \rightarrow (r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})$.

By Rule 5 (Bounded Induction), we first have to prove

(48) $\tau_w \rightarrow F$

which is true since Lemma 19 (Entry) implies $\tau_w \rightarrow \neg run_j$. Let τ be arbitrary fixed with $\tau_w < \tau < \tau_r$ and assume

(49) $\tau_w \rightarrow (F\ \texttt{until}\ \tau)$.

We have to prove

(50) $\tau \rightarrow F$.

We may assume

(51) $\tau \rightarrow \triangle(\beta^j_{11} \vee \beta^j_{12} \vee \exists i(w_i = j \wedge (\alpha^i_{19} \vee \alpha^i_{20} \vee \alpha^i_{21})))$

since these are the only statements that may modify y_j, r_j or run_j.

Case $\tau \to \triangle(\beta^j_{11} \vee \beta^j_{12})$

By $\tau < \tau_e$, Lemma 23 (Not Run) implies

(52) $\tau \to \triangle\neg run_j$.

By the program structure, we thus have

(53) $\tau \to \neg run_j$.

Case $\tau \to \triangle(w_i = j \wedge (\alpha^i_{19} \vee \alpha^i_{20}))$

Lemma 22 (Lock) implies

(54) $\tau \to (\mathtt{elem}_i(j, W_i)\ \mathtt{atprev}\ \alpha^i_7)$.

By Axiom 11 (Wait Statement), we know

(55) $\tau \to (rsem_j\ \mathtt{atprev}\ \alpha^i_7)$.

By Lemma 28 (Result Loop), we thus know

(56) $\tau \to (\neg run_j\ \mathtt{atprev}\ \alpha^i_7)$.

This implies by Lemma 5 (Result Exclusion)

(57) $\tau \to \neg run_j$.

Case $\tau \to \triangle(w_i = j \wedge \alpha^i_{21})$

By the structure of the program and Lemma 5 (Result Exclusion), we know

(58) $\tau \to \triangle(r_j = i \wedge y_j = \overline{x}_i)$.

By Lemma 22 (Lock), we then know

(59) $\tau \to (\mathtt{elem}_i(j, W_i)\ \mathtt{atprev}\ \alpha^i_7)$.

which implies by Lemma 20 (Membership)

(60) $\tau \to (i \in \overline{R}_j\ \mathtt{atprev}\ \alpha^i_7)$

and thus by Lemma 8 (Constancy)

(61) $\tau \to \triangle(i \in \overline{R}_j)$.

This implies by the program structure

(62) $\tau \to (r_j \in \overline{R}_j \wedge y_j = \overline{x}_j)$. $\square$

6.3 Wait Continuation

We are going to prove Proposition 2a of Theorem 1 (Correctness). We assume

(63) $\tau_\Pi \to \exists i \in \overline{R}_j(\diamond\,\mathtt{at}\{\alpha^i_1\})$

and have to prove

(64) $\tau_w \to \diamond\,\mathtt{at}\{\beta^j_e\}$.

By Lemma 31 (Wait Continuation) and Lemma 35 (Result Continuation), we know

(65) $\tau_w \to \diamond\beta^j_{21}$.

We thus may define some time $\tau_e > \tau_w$ such that

(66) $\tau_w \to (\tau_e\ \mathtt{atnext}\ \beta^j_{21})$.

We may assume

(67) $\tau_e \to \neg found_j$

since otherwise, by Lemma 1 (Locality) and the program structure, we are done. Thus we know

(68) $\tau_e \to \Diamond\, \mathtt{at}\{\beta_{24}^j\}$

and have to show

(69) $\tau_e \to (\Diamond run_j \ \mathtt{atnext}\ \beta_{24}^j)$.

From (67) we know by Lemma 26 (Wait End)

(70) $\tau_e \to \forall i \in \overline{R}_j(\mathtt{elem}_i(j, W_i))$

By the program structure, we also know

(71) $\tau_e \to \bigcirc rsem_j$

which implies by Theorem 5 (Sometime)

(72) $\tau_e \to \bigcirc\Diamond run_j$

and, since run_j is nowhere else set to false, (69). $\square$

The following theorem says that, if Π_j has put j into some W_i, some Π_i will eventually set run_j.

Theorem 5 (Sometime) *If j is in some W_i, run_j will be eventually set.*

$$\tau_w \to (((rsem_j \wedge \exists i(\mathtt{elem}_i(j, W_i))) \to \Diamond run_j)\ \mathtt{until}\ \tau_r).$$

Proof: Let $\tau \geq \tau_w$ be arbitrary fixed and assume for some i

(73) $\tau \to (rsem_j \wedge \mathtt{elem}_i(j, W_i))$.

By Lemma 21 (Waiting), we know

(74) $\tau \to \mathtt{at}\{\beta_{22}^j \ldots \beta_e^j\}$.

By Lemma 38 (Result Induction), we know

(75) $\tau_e \to \Diamond(\alpha_4^i \wedge W_i = \emptyset)$.

By (74) and the program structure, we know for some $i' \in \overline{R}_j$

(76) $\tau_e \to \Diamond(\alpha_{16}^{i'} \wedge s_{i'} = i \wedge w_{i'} = j)$.

This implies

(77) $\tau_e \to \Diamond(\alpha_{21}^{i'} \wedge w_{i'} = j)$

and we are by the program structure done. $\square$

The proof of the previous theorem is essentially based on Lemma 38 (Result Induction) saying that $result_i$ always terminates. This lemma holds since $result_i$ defines $value_i$ and sets the flag $avail_i$ i.e. *before* it starts to deliver its result to other tasks. Please note that this is not true in the actual implementation of `pacWait` where $avail_i$ is set *after* the result has been delivered. However, if we add the following axiom that demands a *queue* interpretation of W_i, Theorem 5 (Sometime) is true again.

Axiom 18 (Queue) *W_i is accessed in a first in/first out manner:*

$$\tau_\Pi \to \Box\forall i \forall j \forall j'((\mathtt{elem}_i(j, W_i) \wedge \neg\mathtt{elem}_i(j', W_i)) \to (\neg\mathtt{elem}_i(j, W_i)\ \mathtt{before}\ \neg\mathtt{elem}_i(j', W_i))).$$

By this axiom, we may use Lemma 39 (First Out) to verify Theorem 5 (Sometime):

Proof: Let $\tau \geq \tau_\Pi$ be arbitrary fixed and assume for some i

(78) $\tau \rightarrow (rsem_j \wedge \texttt{elem}_i(j, W_i))$.

By Lemma 21 (Waiting), we know

(79) $\tau \rightarrow \texttt{at}\{\beta^j_{22} \ldots \beta^j_e\}$.

By Lemma 39 (First Out), we know

(80) $\tau \rightarrow \Diamond\neg\texttt{elem}_i(j, W_i)$.

By (79) and the program structure, we know for some $i' \in \overline{R}_j$

(81) $\tau_e \rightarrow \Diamond(\alpha^{i'}_{16} \wedge s_{i'} = i \wedge w_{i'} = j)$.

This implies

(82) $\tau_e \rightarrow \Diamond(\alpha^{i'}_{21} \wedge w_{i'} = j)$

and we are by the program structure done. $\square$

7 Conclusions

We conclude this paper by summarizing the basic proof techniques that we have applied in the program verification. For the semi-formal reasoning process, we usually used the generalized induction rule introduced in this paper.

1. **Locality:** If parts of a program only operate on local variables (i.e. variables that are only modified by the current process), these program parts can be verified without considering the existence of other processes. The complexity of the verification is thus essentially decreased.
2. **Program Variables:** Whenever possible we have expressed program properties in terms of program variables instead of program locations. For instance, Theorem 3 (Found) and Theorem 4 (Awake) are formulated using the variables $found_j$ and run_j and not the program locations β^j_{23} and β^j_{24}, respectively. Thus the propositions become more easy to understand and express the intuitions of the programmer better than by dealing with program locations. This is especially of advantage when code blocks are rearranged such that the locations of statements change but the proposition in terms of program variables still holds.
3. **Critical Regions:** The purpose of a critical region C

   (λ_a) $\quad$ $\mathbf{P}(sem)$
   $\qquad\quad$ C
   (λ_b) $\quad$ $\mathbf{V}(sem)$

   is described by a proposition of the form

   $$\Box(sem \rightarrow A)$$

   saying that A holds when no process is within C and therefore also when a process enters respectively exits C. For proving above formula, we have to prove $\tau_\Pi \rightarrow A$ and, for some arbitrary fixed time $\tau > \tau_\Pi$,

   $$\tau \rightarrow (\lambda_b \rightarrow \bigcirc A)$$

   i.e. that the A holds when C is left. For the proof of this proposition, one may assume the induction hypothesis

   $$\tau \rightarrow (A \texttt{ atprev } \tau_a)$$

i.e. that A was true when C was entered. The reasoning from τ_a to τ_e may be based on the fact that no other process is in C. Furthermore, we have to prove for each statement λ in every other process

$$(\lambda \wedge \mathit{sem} \wedge A) \rightarrow \bigcirc A$$

where we may restrict our consideration to those λ that might change the truth value of A (e.g. by modifying some program variable occurring in A).

4. **While Loops:** The invariant of a loop

$$\begin{array}{ll} (\lambda') & \ldots \\ (\lambda) & \textbf{while } B \textbf{ do} \\ & \quad \ldots \\ (\lambda'') & \quad \ldots \\ & \textbf{endwhile} \end{array}$$

is expressed by a proposition of the form

$$\Box(\,\mathtt{at}\{\lambda\} \rightarrow A)$$

which says that A always holds when execution is at the location λ. For proving this formula, we have to prove

$$\Box(\lambda' \rightarrow \bigcirc A)$$

i.e. that A holds when the loop is entered for the first time and

$$(\,\mathtt{at}\{\lambda\} \wedge \neg\lambda \wedge A) \rightarrow \bigcirc A$$

i.e. A does not become false if some other process is executed (where we may restrict considerations to those statements in other processes that may affect A) and

$$\lambda'' \rightarrow \bigcirc A$$

i.e. A holds after each iteration. For proving this fact, we may apply the induction hypothesis

$$\lambda'' \rightarrow (A \wedge B \ \mathtt{atprev}\ \lambda)$$

and then reason from one execution of λ to the next.

5. **Nested Loops:** The invariants A_1 and A_2 of two nested loops

$$\begin{array}{ll} (\lambda_1) & \textbf{while } B_1 \textbf{ do} \\ & \quad \ldots \\ (\lambda_2) & \quad \textbf{while } B_2 \textbf{ do} \\ & \quad\quad \ldots \\ & \quad \textbf{endwhile} \\ & \quad \ldots \\ & \textbf{endwhile} \end{array}$$

can be only verified by forming a joint invariant

$$A \equiv (\,\mathtt{at}\{\lambda_1\} \rightarrow A_1) \wedge (\,\mathtt{at}\{\lambda_2\} \rightarrow A_2)$$

and proving this joint proposition at all interesting transitions. A may be assumed for all former time moments (similar to the case of a single while loop).

6. **Critical Loops:** If the entry of a loop is part of a critical region

```
        P(sem)
(λ)     while B do
          ...
          V(sem)
          ...
          P(sem)
        endwhile
        V(sem)
```

the loop invariant actually reduces to the invariant of the critical region protected by *sem* extended by the proposition B i.e.

$$\Box(sem \wedge B \rightarrow A)$$

and then proceeds in a way similar to simple while loops.

7. **Continuation:** A good strategy to verify the continuation of a program is to verify the freedom of deadlocks of critical regions and the termination of loops from inside to the outside starting with the innermost region/loop. Parallel programs are usually constructed in such a way that there exists a partial ordering $\prec$ on semaphores such that $s_1 \prec s_2$ implies that a critical region protected by s_1 is only entered if the process is not in a critical region protected by s_2. This ordering also guides the structure of the proof.

There can be no general recipe how the verification of a parallel program with shared variables has to be organized to be manageable but also convincing. In the worst case, all parallel processes may be coupled so tightly that the full calculus of temporal logic has to be applied by constructing for every statement in every process an assertion that describes the full state of the program. Each of these assertions has then to be verified against every statement.

However, such situations probably occur very rarely. In many cases, the verification style presented in this paper may save a considerable amount of work and make the verification a valuable documentation of the program. A high-level parallel programming model should be constructed in such a way that the model is powerful enough to express the parallel algorithms one is interested in but also restricted enough to cut down the complexity of parallel program verification.

References

[CM88] Mani K. Chandy and Jayadev Misra. *Parallel Program Design — A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1985.

[HSN+92] Hoon Hong, Wolfgang Schreiner, Andreas Neubacher, Kurt Siegl, Hans-Wolfgang Loidl, Tudor Jebelean, and Peter Zettler. PACLIB User Manual. Technical Report 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992. Also: Technical Report 92-9, ACPC Technical Report Series, Austrian Center for Parallel Computation, July 1992.

[Krö87] Fred Kröger. *Temporal Logic of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science.* Springer, Berlin, 1987.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer, 1980.

[Mil93] Robert Milner. Elements of Interaction. *Communications of the ACM*, 36(1):78–89, January 1993.

[MP81] Z. Manna and Z. Pnueli. Verification of Concurrent Programs, Part 1: The Temporal Framework. Technical Report STAN-CS-81-836, Stanford University, Palo Alto, CA, June 1981.

[OG76] S. Owicki and D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[OL82] S. Owicki and L. Lamport. Proving Liveness Properties of Parallel Programs: An Axiomatic Approach. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[Oss83] Marty Ossefort. Correctness Proofs of Communicating Processes: Three Illustrative Examples from the Literature. *ACM Transactions on Programming Languages and Systems*, 5(4):620–640, October 1983.

[Sch92] Wolfgang Schreiner. The Design of the PACLIB Kernel. Technical Report 92-33, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

A The PACLIB Kernel

```
/***************************************************************************/
void pacShell(pacFunction f,
  pacObject a1, pacObject a2, pacObject a3, pacObject a4,
  pacObject a5, pacObject a6, pacObject a7, pacObject a8) {
/***************************************************************************/

  pacResult(f(a1, a2, a3, a4, a5, a6, a7, a8));
  pacDie();

}

/***************************************************************************/
void pacResult(pacObject v) {
/***************************************************************************/

  pacObject value, value0;             /* function results */
  pacTask result, result0;             /* result descriptor */
  pacTaskList rlist;                   /* list of result descriptors */
  pacTaskList tlist, tlist0, tlist1;   /* lists of task descriptors*/
  uTask task, task0;                   /* task descriptors */
  int deliver;                         /* deliver flag */

  result = uWorkTask->pac.result;      /* result descriptor of current task */

  resLock(result);                     /* lock descriptor */
  resGet(result, &task0, &value, &tlist); /* get associated task list */
  while (!listNull(tlist)) {           /* there are some waiting tasks */
    task = (uTask)listHead(tlist);     /* get next waiting task */
    resUnlock(result);                 /* unlock result descriptor */

    taskLock(task);                    /* lock task descriptor */

    resLock(result);                   /* lock descriptor again */
    resGet(result, &task0, &value, &tlist);
    deliver = (!listNull(tlist)) && ((uTask)listHead(tlist) == task);
    resUnlock(result);                 /* unlock descriptor again */

    if (deliver) {                     /* remove task from result descr */

      rlist = task->pac.wait;          /* results the task waited for */
      while (!listNull(rlist)) {       /* remove task from result descs */
        result0 = listHead(rlist);     /* get next result descriptor */
        resLock(result0);              /* lock result descriptor */
        resGet(result0, &task0, &value0, &tlist0);
        tlist1 = listNew();            /* empty predecessor list */
        while (task != (uTask)listHead(tlist0)) { /* search for task */
          tlist1 = tlist0;             /* new predecessor */
          tlist0 = listTail(tlist0);   /* task descriptor not yet found */
        }
        if (listNull(tlist1))          /* head of queue must be updated */
          resSetWait(result0, listTail(tlist0));
        else                           /* unlink the task from the queue */
          listSetTail(tlist1, listTail(tlist0));
        if (value0 == tlist0)          /* last task in list removed */
          resSetValue(result0, tlist1);
        resUnlock(result0);            /* unlock result descriptor */
        rlist = listTail(rlist);       /* get next descriptor */
```

```
      }

      task->pac.wait = result;        /* deliver result descriptor */
      task->pac.value = v;            /* deliver actual result value */
      uWakeTask(task);                /* wake up task */

    }

    taskUnlock(task);                 /* unlock this task */

    resLock(result);                  /* lock result descriptor */
    resGet(result, &task0, &value, &tlist); /* get associated task list */
  }

  /* deliver result */
  resSetTask(result, AVAILABLE);      /* no task is executing any more */
  resSetValue(result, v);             /* this is the result value */
  resSetWait(result, NIL);            /* no tasks waiting any more */
  resUnlock(result);                  /* unlock result descriptor */

}

/****************************************************************************/
pacObject pacWaitList(pacTask *tp, pacTaskList rlist) {
/****************************************************************************/

  pacTaskList rlist0;                 /* temp. list of result descriptors */
  pacTask result;                     /* actual result descriptor */
  uTask task;                         /* actual usystem task */
  pacObject value, v;                 /* actual result values */
  pacTaskList tlist, tlist0;          /* lists of waiting tasks */
  pacObject new;                      /* new list cell */
  int found = FALSE;                  /* value was found */

  taskLock(uWorkTask);                /* lock this task */
  rlist0 = rlist;                     /* list of result descriptors */

  while (!listNull(rlist0) && !found) { /* still more tasks to wait for */

    result = listHead(rlist0);        /* next result descriptor */
    resLock(result);                  /* lock descriptor and get values */
    resGet(result, &task, &value, &tlist);

    if (!taskAvail(task)) {           /* result not available */
      new = listCons((pacObject)uWorkTask, listNew());
      resSetValue(result, new);       /* tail of waiting queue */
      if (listNull(value))            /* no waiting tasks yet */
        resSetWait(result, new);      /* head of waiting queue */
      else                            /* there are already waiting tasks */
        listSetTail(value, new);      /* update tail of waiting queue */
      resUnlock(result);              /* unlock result descriptor */
      rlist0 = listTail(rlist0);      /* get rest of result descriptor list */
      continue;                       /* continue next loop iteration */
    }

    /* task has already terminated and delivered result */
    resUnlock(result);                /* unlock descriptor */
    if (tp) *tp = result;             /* remember result descriptor */
    v = value;                        /* result value */
```

```
    found = TRUE;                      /* value was found */

    while(rlist != rlist0) {           /* remove task from descriptors again */
      result = listHead(rlist);        /* next result descriptor */
      resLock(result);                 /* lock descriptor */
      resGet(result, &task, &value, &tlist);
      tlist0 = listNew();              /* predecessor in list */
      while ((uTask)listHead(tlist) != uWorkTask) {
        tlist0 = tlist;                /* new predecessor */
        tlist = listTail(tlist);       /* next list tail */
      }
      if (listNull(tlist0))            /* no predecessor in list */
        resSetWait(result, listTail(tlist));
      else                             /* predecessor in list */
        listSetTail(tlist0, listTail(tlist));
      if (value == tlist) resSetValue(result, tlist0);
      resUnlock(result);               /* unlock result descriptor */
      rlist = listTail(rlist);         /* rest of list */
    }

  }

  if (found)                           /* value was found */
    taskUnlock(uWorkTask);             /* unlock current task */
  else {                               /* value not found */
    uWorkTask->pac.wait = rlist;       /* list of results that we wait for */
    taskUnlock(uWorkTask);             /* unlock current task */
    uBlockTask();                      /* block current task */
    if (tp) *tp = uWorkTask->pac.wait; /* this is the result descriptor */
    v = uWorkTask->pac.value;          /* this is the actual result */
    uWorkTask->pac.wait = listNew();   /* reset wait field */
    uWorkTask->pac.value = listNew();  /* reset value field */
  }

  return(v);                           /* return result */

}
```

B Lemmas

We now list all auxiliary lemmas classified into the following categories:

1. **Mutual Exclusion Lemmas** restricting the access to program regions and data structures.
2. **Operation Lemmas** describing the validity and the effects of several program operations.
3. **Structure Lemmas** describing properties that can be derived from the static structure of the program.
4. **Main Lemmas** that essentially capture the dynamic behavior of the program.
5. **Continuation Lemmas** that describe eventuality properties of the program.

The lemmas are listed in a bottom-up order i.e. the proof of a lemma is based only on previously listed lemmas (with some explicitly mentioned exceptions).

B.1 Mutual Exclusion Lemmas

We define the predicate $\mathtt{invar}(a)$ by

$$\mathtt{invar}(a) \equiv \forall \overline{a}(a = \overline{a} \rightarrow \bigcirc a = \overline{a})$$

saying "a is not changed in the current time moment" and the propositional operator $\mathtt{exor}(\{A_1, \ldots, A_n\})$

$$\begin{aligned} \mathtt{exor}(\{A_1, \ldots, A_n\}) \equiv\ & (A_1 \wedge \neg A_2 \wedge \ldots \wedge \neg A_n) \vee \\ & (\neg A_1 \wedge A_2 \wedge \ldots \wedge \neg A_n) \vee \\ & \ldots \\ & (\neg A_1 \wedge \neg A_2 \wedge \ldots \wedge A_n) \end{aligned}$$

saying "exactly one of $A_1, \ldots, A_n$ is true".

The following lemma declares a set of program variables to be "local".

Lemma 1 (Locality) *The variables w_i, S_i, s_i, $value_i$, $avail_i$, $deliver_i$, P_j, p_j, Q_j, q_j and $found_j$ are only modified by the processes i and j, respectively:*

$$\begin{aligned} \tau_\Pi \rightarrow\ & \Box \forall i \forall i' \forall l ((\lambda_l^{i'} \wedge i' \neq i) \rightarrow \\ & (\mathtt{invar}(w_i) \ \wedge\ \mathtt{invar}(S_i) \ \wedge\ \mathtt{invar}(s_i) \ \wedge \\ & \mathtt{invar}(value_i) \ \wedge\ \mathtt{invar}(avail_i) \ \wedge\ \mathtt{invar}(deliver_i))) \end{aligned}$$

$$\begin{aligned} \tau_\Pi \rightarrow\ & \Box \forall j \forall j' \forall l ((\lambda_l^{j'} \wedge j' \neq j) \rightarrow \\ & (\mathtt{invar}(P_j) \ \wedge\ \mathtt{invar}(p_j) \ \wedge\ \mathtt{invar}(Q_j) \ \wedge\ \mathtt{invar}(q_j) \ \wedge\ \mathtt{invar}(found_j))) \end{aligned}$$

Proof: These variables are only modified in $result_i$ and $wait_j$, respectively. $\Box$

The following lemma declares the mutual exclusion of several critical regions protected by the semaphore $wsem_i$. If $wsem_i$ is true, no process is within the denoted regions; if $wsem_i$ is false, exactly one process is in one of these regions. Please not that only one critical regions is "statically defined" by the structure of the program (i.e. by the value of the location counter of a process), while the others depend on the contents of the program variables $s_{i'}$ and p_j, respectively.

Lemma 2 (Wait Semaphore) *The family of semaphores $wsem_i$ establishes the mutual exclusion of several critical regions:*

$$\begin{aligned}
\tau_\Pi \to \Box\forall i(&\mathtt{exor}\{\\
&wsem_i,\\
&\mathtt{at}\{\alpha_4^i \ldots \alpha_6^i, \alpha_9^i, \alpha_{10}^i, \alpha_{24}^i\},\\
&s_1 = i \wedge \mathtt{at}\{\alpha_{16}^1, \alpha_{17}^1\},\\
&\ldots\\
&s_n = i \wedge \mathtt{at}\{\alpha_{16}^n, \alpha_{17}^n\},\\
&p_1 = i \wedge \mathtt{at}\{\beta_6^1 \ldots \beta_8^i, \beta_{10}^i\},\\
&\ldots\\
&p_n = i \wedge \mathtt{at}\{\beta_6^n \ldots \beta_8^n, \beta_{10}^n\},\\
&q_1 = i \wedge \mathtt{at}\{\beta_{18}^1 \ldots \beta_{19}^1\},\\
&\ldots\\
&q_n = i \wedge \mathtt{at}\{\beta_{18}^n \ldots \beta_{19}^n\}\}).
\end{aligned}$$

Proof: Proving this "obvious" program property is a basically simple but practically tedious task. By Lemma 2 (Invariant Rule), it suffices to show that mutual exclusion holds initially (which is a consequence of start_Π) and that it is invariant under each action λ.

Since s_i, p_j and q_j are local to process i and j respectively and s_i, p_j and q_j are not changed within the critical regions, it suffices to check the invariance of the property at the entries and the exits of the critical regions:

1. $\lambda \in \{\alpha_3^i, \alpha_6^i, \alpha_8^i, \alpha_{10}^i, \alpha_{23}^i, \alpha_{24}^i\}$,
2. $\lambda \in \{\alpha_{15}^{i'}, \alpha_{17}^{i'}\}$ assuming $s_{i'} = i$,
3. $\lambda \in \{\beta_5^j, \beta_8^j, \beta_{10}^j\}$ assuming $p_j = i$ and
4. $\lambda \in \{\beta_{17}^j, \beta_{19}^j\}$ assuming $p_j = i$.

The details are left to the reader. □

The previous lemma only describes the low-level effect of the semaphore $wsem_i$; its high-level purpose (the protection of the set W_i) is actually expressed by the following lemma.

Lemma 3 (Wait Exclusion) *Each wait set W_i is only modified in a critical region protected by $wsem_i$:*

$$\tau_\Pi \to \Box\forall i(wsem_i \to \mathtt{invar}(W_i)).$$

Proof: This is a corollary of Lemma 2 (Wait Semaphore) (check!). □

Likewise, we define the effect and the purpose of the semaphore $rsem_j$.

Lemma 4 (Result Semaphore) *The family of semaphores $rsem_j$ establishes the mutual exclusion of several critical regions:*

$$\begin{aligned}
\tau_\Pi \to \Box\forall j(&\mathtt{exor}\{\\
&rsem_j,\\
&\mathtt{at}\{\beta_3^j \ldots \beta_{21}^j\},\\
&w_1 = j \wedge \mathtt{at}\{\alpha_8^1 \ldots \alpha_{22}^1\},\\
&\ldots\\
&w_n = j \wedge \mathtt{at}\{\alpha_8^n \ldots \alpha_{22}^n\}\}).
\end{aligned}$$

Proof: The proof proceeds analogously to the proof of Lemma 2 (Wait Semaphore) by checking that mutual exclusion holds initially (which is a direct consequence of A) and that this property is invariant under the following actions λ:

1. $\lambda \in \{\beta_2^j, \beta_{21}^j\}$ and
2. $\lambda \in \{\alpha_7^i, \alpha_{22}^i\}$ (assuming $w_i = j$).

The details are left to the reader. □

Lemma 5 (Result Exclusion) *The following actions are only performed within a critical region protected by $rsem_j$:*

1. *y_j is modified.*
2. *r_j is modified.*
3. *run_j is set to true.*
4. *j is added to or removed from some W_i.*

Formally,

$$\begin{aligned}&\tau_\Pi \rightarrow \Box(rsem_j \rightarrow \\ &\quad \mathtt{invar}(y_j) \ \wedge \ \mathtt{invar}(r_j) \ \wedge \mathtt{invar}(run_j) \ \wedge \\ &\quad \forall i(\mathtt{elem}_i(j, W_i) \leftrightarrow \bigcirc \mathtt{elem}_i(j, W_i))).\end{aligned}$$

Proof: This is a corollary of Lemma 4 (Result Semaphore), Lemma 2 (Wait Semaphore) and the program structure. $\Box$

B.2 Operation Lemmas

The effect of the operations **add** and **rem** on W_i is described by the following lemma.

Lemma 6 (Set Update) *If W_i is updated by adding or removing some j, then the update has achieved a consistent state at the end of the corresponding critical region:*

$$\begin{aligned}&\Box\forall i \forall j \forall k \forall \overline{W}(\\ &\quad ((\alpha^i_{15} \wedge W_i = \overline{W} \wedge w_i = j) \rightarrow (W_i = \overline{W} \backslash \{j\} \ \mathtt{atnext} \ \alpha^i_{17})) \ \wedge \\ &\quad ((\beta^j_5 \wedge p_j = k \wedge W_k = \overline{W} \wedge \neg avail_k) \rightarrow (W_k = \overline{W} \cup \{j\} \ \mathtt{atnext} \ \beta^j_8)) \ \wedge \\ &\quad ((\beta^j_{17} \wedge q_j = k \wedge W_k = \overline{W}) \rightarrow (W_k = \overline{W} \backslash \{j\} \ \mathtt{atnext} \ \beta^j_{19}))).\end{aligned}$$

Proof: By Lemma 2 (Wait Semaphore). $\Box$

The following lemma guarantees that we are allowed to use the $\in$ assignment in the program model.

Lemma 7 (Element) *The $\in$ assignment is well-defined:*

$$\begin{aligned}&\tau_\Pi \rightarrow \Box\forall i((\alpha^i_5 \rightarrow W_i \neq \emptyset) \wedge (\alpha^i_{14} \rightarrow S_i \neq \emptyset)), \\ &\tau_\Pi \rightarrow \Box((\beta^j_4 \rightarrow P_j \neq \emptyset) \wedge (\beta^j_{16} \rightarrow Q_j \neq \emptyset)).\end{aligned}$$

Proof: Lemma 2 (Wait Semaphore) and Axiom 14 (While Loop) imply

$$\tau_\Pi \rightarrow \Box\forall i(\alpha^i_5 \rightarrow W_i \neq \emptyset).$$

The rest is a consequence of Lemma 1 (Locality) and Axiom 14 (While Loop). $\Box$

B.3 Structure Lemmas

The following lemma tells us that in certain contexts we need not distinguish between x_i and $\bar{x}_i$ and between R_j and $\bar{R}_j$:

Lemma 8 (Constancy) *The parameters x_i and R_j are not changed in $result_i$ and $wait_j$, respectively:*

$$\begin{aligned}&\tau_\Pi \rightarrow \Box\forall i(\alpha^i_1 \rightarrow \Box(x_i = \overline{x}_i)) \ \wedge \\ &\tau_w \rightarrow ((R_j = \overline{R}_j \ \mathtt{until} \ \tau_r) \wedge (R_j = \overline{R}_j \ \mathtt{atnext} \ \tau_r)).\end{aligned}$$

Proof: By the structure of the program. $\Box$

The following lemmas describe the state of several program variables at certain program locations.

Lemma 9 (Available) *After the available flag is set, it remains true in $result_i$:*

$$\tau_\Pi \rightarrow \Box\forall i(\,\mathtt{at}\{\alpha_3^i \ldots \alpha_e^i\} \rightarrow avail_i).$$

Proof: By Lemma 1 (Locality). $\Box$

Lemma 10 (If Available) *In the else branch of $wait_j$, $avail_{p_j}$ always holds:*

$$\tau_\Pi \rightarrow \Box\forall j(\,\mathtt{at}\{\beta_{10}^j \ldots \beta_{20}^j\} \rightarrow avail_{p_j}).$$

Proof: By Axiom 12 (Two Branch Conditional) and Lemma 1 (Locality). $\Box$

Lemma 11 (Not Available) *If process i does not deliver a result, $avail_i$ will never hold:*

$$\tau_\Pi \rightarrow \forall i(\Box\neg\,\mathtt{at}\{\alpha_1^i\} \rightarrow \Box\neg avail_i).$$

Proof: By Rule 4 (Induction). $\Box$

Lemma 12 (Process Set) *Within $wait_j$, the variables p_j and r_j are always members of the process set $\overline{R}_j$:*

$$\tau_\Pi \rightarrow \Box((\,\mathtt{at}\{\beta_5^j \ldots \beta_{20}^j\} \rightarrow p_j \in \overline{R}_j) \wedge (\,\mathtt{at}\{\beta_{17}^j \ldots \beta_{20}^j\} \rightarrow q_j \in \overline{R}_j).$$

Proof: By Rule 4 (Induction). $\Box$

Lemma 13 (Result) *If the available flag is set, then the result is indeed available:*

$$\tau_\Pi \rightarrow \Box\forall i(avail_i \rightarrow value_i = \overline{x}_i).$$

Proof: By Rule 8 (Constancy) using that only α_2^i sets $avail_i$ to true. $\Box$

Lemma 14 (Found) *After $found_j$ is set to true, it remains true:*

$$\tau_\Pi \rightarrow \Box(\,\mathtt{at}\{\beta_{14}^j \ldots \beta_{20}^j\} \rightarrow found_j).$$

Proof: By Lemma 1 (Locality) and the structure of t eh program. $\Box$

Lemma 15 (Not Found) *Before $found_j$ is set to true, $found_j$ is false:*

$$\tau_\Pi \rightarrow \Box(\,\mathtt{at}\{\beta_4^j \ldots \beta_{13}^j\} \rightarrow \neg found_j).$$

Proof: By Axiom 14 (While Loop) and Lemma 1 (Locality). $\Box$

Lemma 16 (Reset) *If $found_j$ is set to false, it was true before:*

$$\tau_\Pi \rightarrow \Box(\,\mathtt{at}\{\beta_{23}^j\} \rightarrow found_j).$$

Proof: By Axiom 12 (Two Branch Conditional) and Lemma 1 (Locality). $\Box$

Lemma 17 (Sleeping) *If j is going to sleep, $found_j$ is false.*

$$\tau_\Pi \rightarrow \Box(\,\mathtt{at}\{\beta_{24}^j\} \rightarrow \neg found_j).$$

Proof: By Axiom 12 (Two Branch Conditional) and Lemma 1 (Locality). $\Box$

Lemma 18 (Deliver) *If process i is in the "then" branch of $result_i$, then $deliver_i$ was set at the previous α_9^i:*

$$\tau_\Pi \rightarrow \Box\forall i(\,\mathtt{at}\{\alpha_{12}^i \ldots \alpha_{21}^i\} \rightarrow (deliver_i \wedge w_i \in W_i \;\mathtt{atprev}\; \alpha_9^i)).$$

Proof: This is a consequence of Axiom 13 (One Branch Conditional) and Lemma 1 (Locality). $\Box$

B.4 Main Lemmas

In this section, we give the key lemmas that capture the dynamic behavior of the program. The organization of these lemmas is as follows:

1. **Entry:** Lemma 19 describes the state of Π when $wait_j$ is entered (i.e. it is of the form $\tau_w \to A$).
2. **Others:** Most lemmas capture the behavior of Π during the execution of $wait_j$ (i.e. they are of the form $\tau_w \to (A \texttt{ until } \tau_r)$).
3. **Exit:** Lemma 30 describes the state of Π immediately before $wait_j$ is left (i.e. it is of the form $\tau_f \to (A \texttt{ until } \tau_r)$.

For proving these lemmas, we perform an induction over the number of times that $wait_j$ has been executed before:

1. **Induction Base:** We prove that Lemma 19 holds when $wait_j$ is entered for the first time i.e. has not been executed before.
2. **Others:** Based on the induction hypothesis Lemma 19, we prove that the other lemmas hold.
3. **Exit:** Based on the other lemmas, we prove that Lemma 30 holds.
4. **Induction Step:** based on Lemma 30, we prove that Lemma 19 holds when $wait_j$ is entered for the next time.

Both the induction base and the induction step are given in the proof of Lemma 19 (Entry) while the other proofs are listed after the corresponding lemmas. Please note that the use of Lemma 19 in the Lemma 30 is therefore not a vicious circle but a valid assumption in the induction step.

Lemma 19 (Entry) *When $wait_j$ is entered, $found_j$ and run_j are false and j is not in any wait set W_i and no process i is going to deliver its result to j:*

$$t_w \to (\neg found_j \wedge \neg run_j \wedge \forall i(\neg \texttt{elem}_i(j, W_i) \wedge (\texttt{at}\{\alpha^i_{12} \dots \alpha^i_{21}\} \to w_i \neq j))).$$

Proof: We will prove this lemma by induction over the number of times that $wait_j$ has been executed. Define

(1) $B \equiv \neg found_j \wedge \neg run_j \wedge \forall i(\neg \texttt{elem}_i(j, W_i) \wedge (\texttt{at}\{\alpha^i_{12} \dots \alpha^i_{21}\} \to w_i \neq j))$.

We will prove

(2) $\tau_w \to B$.

Induction Base

We assume that $wait_j$ is entered for the first time i.e.

(3) $\tau_\Pi \to (\tau_w \texttt{ atnext } \beta^j_1)$.

By $start_\Pi$, we know

(4) $\tau_\Pi \to B$.

Since only $wait_j$ sets $found_j$ to true and extends W_i by j, we therefore know

(5) $\tau_w \to (\neg found_j \wedge \forall i(\neg \texttt{elem}_i(j, W_i)))$

and also

(6) $\tau_\Pi \to ((\neg found_j \wedge \forall i(\neg \texttt{elem}_i(j, W_i))) \texttt{ until } \tau_w)$.

From this, we know

(7) $\tau_\Pi \to (\forall i(\, \mathtt{at}\{\alpha^i_{12} \ldots \alpha^i_{21}\} \to w_i \neq j)\ \mathtt{until}\ \tau_w)$

which implies

(8) $\tau_w \to (\neg run_j \wedge \forall i(\, \mathtt{at}\{\alpha^i_{12} \ldots \alpha^i_{21}\} \to w_i \neq j))$.

Induction Step

Let τ'_w such that

(9) $\tau_w \to (\tau'_w\ \mathtt{atnext}\ \beta^j_1)$.

From Lemma 30 (Exit), we know

(10) $\tau_r \to B$.

Since only $wait_j$ sets $found_j$ to true and extends W_i by j, it is easy to derive

(11) $\tau'_w \to B$.

Therefore we are done. □

The following lemma establishes a relationship between W_i and R_j.

Lemma 20 (Membership) *If process j is in the wait set W_i, then process i is in the process set $\overline{R}_j$ and j executed $wait_j$:*

$$\tau_w \to \forall i(\mathtt{elem}_i(j, W_i) \to i \in \overline{R}_j)\ \mathtt{until}\ \tau_r.$$

Proof: Let i be arbitrary fixed. We know from Lemma 19 (Entry)

(1) $\tau_w \to \neg\mathtt{elem}_i(j, W_i)$.

Now let τ be arbitrary fixed with $\tau_w < \tau < \tau_r$ such that

(2) $\tau \to \mathtt{elem}_i(j, W_i)$.

By the program structure, we know that for some τ_p with $\tau_r < \tau_p < \tau$

(3) $\tau_p \to \beta^j_7 \wedge p_j = i$.

By Lemma 12 (Process Set), we then know

(4) $\tau_p \to i \in \overline{R}_j$

and by Lemma 8 (Constancy) we are done. □

The following lemma will be used to guarantee that no Π_i delivers a result to Π_j "too early".

Lemma 21 (Waiting) *If outside the critical region protected by $rsem_j$ j is in some wait set W_i, then process j is ready to receive a result.*

$$\tau_w \to ((rsem_j \wedge \mathtt{elem}_i(j, W_i)) \to \mathtt{at}\{\beta^j_{22}, \beta^j_e\})\ \mathtt{until}\ \tau_r$$

Proof: Let $\tau \geq \tau_w$ be arbitrary fixed such that

(1) $\tau \to (rsem_j \wedge \mathtt{elem}_i(j, W_i))$.

By Lemma 19 (Entry), we know

(2) $\tau_w \to \forall i(\neg\mathtt{elem}_i(j, W_i))\ \mathtt{until}\ \tau_d$.

By Lemma 4 (Result Semaphore), we know

(3) $\tau_d \to \neg rsem_j\ \mathtt{until}\ \tau_e$.

We therefore know $\tau > \tau_d$, and thus

(4) $\tau_d \to \mathtt{at}\{\beta^j_{22} \ldots \beta^j_e\}$. □

The next lemma fixes a non-obvious property at the start of the critical region protected by $rsem_j$.

Lemma 22 (Lock) *If i is going to deliver its result to j then j was in the wait set of i when i locked the critical region protected by $rsem_j$:*

$$\tau_\Pi \to \Box\forall i((\,\texttt{at}\{\alpha^i_{12}\ldots\alpha^i_{21}\} \wedge w_i = j) \to (\texttt{elem}_i(j, W_i)\ \texttt{atprev}\ \alpha^i_7)).$$

Proof: Let i be arbitrary fixed and define

(1) $D \equiv \forall i((\,\texttt{at}\{\alpha^i_{12}\ldots\alpha^i_{21}\} \wedge w_i = j) \to (\texttt{elem}_i(j, W_i)\ \texttt{atprev}\ \alpha^i_7))$.

From $start_\Pi$, we know

(2) $\tau_\Pi \to D$.

Let time $\tau > \tau_\Pi$ be arbitrary fixed such that

(3) $\tau \to (\,\texttt{at}\{\alpha^i_{12}\ldots\alpha^i_{21}\} \wedge w_i = j)$.

By Lemma 18 (Deliver), we know

(4) $\tau \to (j \in W_i\ \texttt{atprev}\ \alpha^i_9)$.

By Lemma 2 (Wait Semaphore), we then know

(5) $\tau \to (j \in W_i\ \texttt{atprev}\ \alpha^i_8)$.

By Lemma 5 (Result Exclusion), we then know

(6) $\tau \to (\texttt{elem}_i(j, W_i)\ \texttt{atprev}\ \alpha^i_7)$. □

The following lemma guarantees that no Π_i delivers a result to Π_j before Π_j has left the critical region protected by $rsem_j$.

Lemma 23 (Not Run) *After the entry of $wait_j$, run_j remains false until the end of the critical region protected by $rsem_j$:*

$$\tau_w \to (\neg run_j\ \texttt{until}\ \tau_e).$$

Proof: By Lemma 19 (Entry), we know

(1) $\tau_w \to \neg run_j$.

By Lemma 19 (Entry) and the program structure, we also know

(2) $\tau_w \to (\forall i(\neg\texttt{elem}_i(j, W_i))\ \texttt{until}\ \beta^j_3)$.

By Axiom 11 (Wait Statement), we know

(3) $\tau_w \to (\neg rsem_j\ \texttt{atnext}\ \beta^j_3)$.

Moreover, we know by Lemma 22 (Lock) and Axiom 11 (Wait Statement)

(4) $\tau_w \to ((rsem_j \wedge \texttt{elem}_i(j, W_i))\ \texttt{before}\ (\alpha^i_{21} \wedge w_i = j))$.

By Lemma 4 (Result Semaphore), (2), (3) and (4) imply

(5) $\tau_w \to (\neg run_j\ \texttt{atnext}\ \beta^j_2)$.

By Lemma 5 (Result Exclusion), we are done. □

The next both lemmas describe the invariants that hold at each iteration of the inner respectively outer loop of $wait_j$. Unfortunately, using temporal logic the invariants of nested loops cannot be proved separately since they mutually depend on each other. Therefore, we combine them to one and verify the correctness of the joint proposition

- at the first entry of each loop,
- at the transition to the next iteration of each loop,
- at the exit of each loop,

- at the entry of a loop when another process is executed.

This yields seven cases for both loops (since the exit of the inner loop coincidences with the entry of the outer loop).

Lemma 24 (Remove Loop) *At the start of each iteration of the inner loop of $wait_j$, j is only in the wait sets W_i of all $i \in Q_j$:*

$$\begin{aligned}\tau_w \to ((\ \mathtt{at}\{\beta^j_{15}\} \to \\ &\forall i \in Q_j(\mathtt{elem}_i(j, W_i))\wedge \\ &\forall i \in \overline{R}_j \backslash Q_j(\neg\mathtt{elem}_i(j, W_i)))\ \mathtt{until}\ \tau_r).\end{aligned}$$

Proof: The proof is contained in the proof of Lemma 25 (Wait Loop). □

Lemma 25 (Wait Loop) *At the start of each iteration of the outer loop of $wait_j$, the following invariant holds:*

1. *If $found_j$ then j is not in any wait set W_i.*
2. *If $\neg found_j$ then j is only in the wait sets W_i of all $i \in \overline{R}_j \backslash P_j$.*

Formally,

$$\begin{aligned}\tau_w \to ((\ \mathtt{at}\{\beta^j_3\} \to \\ &(found_j \wedge \forall i \in \overline{R}_j(\neg\mathtt{elem}_i(j, W_i)))\ \vee \\ &(\neg found_j \wedge \forall i \in P_j(\neg\mathtt{elem}_i(j, W_i)) \wedge \forall i \in \overline{R}_j \backslash P_j(\mathtt{elem}_i(j, W_i)))\ \mathtt{until}\ \tau_r).\end{aligned}$$

Proof: We will prove this lemma simultaneously with Lemma 24 (Remove Loop). Define

(1) $E \equiv found_j \wedge \forall i \in \overline{R}_j(\neg\mathtt{elem}_i(j, W_i))$

(2) $F \equiv \neg found_j \wedge \forall i \in P_j(\neg\mathtt{elem}_i(j, W_i)) \wedge \forall i \in \overline{R}_j \backslash P_j(\mathtt{elem}_i(j, W_i))$

(3) $G \equiv \forall i \in Q_j(\mathtt{elem}_i(j, W_i)) \wedge \forall i \in \overline{R}_j \backslash Q_j(\neg\mathtt{elem}_i(j, W_i))$

(4) $H \equiv (\ \mathtt{at}\{\beta^j_3\} \to (E \vee F)) \wedge (\ \mathtt{at}\{\beta^j_{15}\} \to G)$

We will prove

(5) $\tau_w \to (H\ \mathtt{until}\ \tau_r)$.

First we have to prove

(6) $\tau_w \to H$

which is obviously true. Second let τ be arbitrary fixed with $\tau_w < \tau < \tau_r$ and assume

(7) $\tau_w \to (H\ \mathtt{until}\ \tau)$.

We have to prove

(8) $\tau \to H$.

We may restrict our consideration to the following cases

(9) $\tau \to \triangle(\beta^j_2 \vee (\ \mathtt{at}\{\beta^j_3\} \wedge \neg\beta^j_3) \vee \beta^j_9 \vee \beta^j_{14}) \vee$

(10) $\tau \to \triangle((\ \mathtt{at}\{\beta^j_{15}\} \wedge \neg\beta^j_{15}) \vee (\beta^j_{15} \wedge Q_j = \emptyset) \vee \beta^j_{20})$

since otherwise $\tau \to \neg\ \mathtt{at}\{\beta^j_3, \beta^j_{15}\}$.

Case $\tau \to \triangle\beta^{\mathbf{j}}_{\mathbf{2}}$

By the case assumption, we know

(11) $\tau \to \mathtt{at}\{\beta^j_3\}$.

By Lemma 19 (Entry) and Lemma 1 (Locality), we know

(12) $\tau \to (\neg found_j \wedge P_j = \overline{R}_j \wedge \forall i \in \overline{R}_j(\neg\mathtt{elem}_i(j, W_i)))$

and thus

(13) $\tau \rightarrow F$.

Case $\tau \rightarrow \triangle(\texttt{at}\{\beta_3^j\} \wedge \neg\beta_3^j)$

By the case assumption, we know

(14) $\tau \rightarrow \texttt{at}\{\beta_3^j\}$.

By the induction hypothesis (7), we also know

(15) $\tau \rightarrow \triangle(E \vee F)$.

By Lemma 1 (Locality) and Lemma 5 (Result Exclusion), we then know

(16) $\tau \rightarrow E \vee F$.

Case $\tau \rightarrow \triangle\beta_9^j$

By the case assumption, we know

(17) $\tau \rightarrow \texttt{at}\{\beta_3^j\}$.

By the induction hypothesis (7) and Axiom 14 (While Loop), we know

(18) $\tau \rightarrow ((H \wedge P_j \neq \emptyset \wedge \neg found_j)\ \texttt{atprev}\ \beta_3^j)$

which implies

(19) $\tau \rightarrow ((F \wedge P_j \neq \emptyset)\ \texttt{atprev}\ \beta_3^j)$.

By the case assumption, Lemma 5 (Result Exclusion) and Lemma 8 (Constancy), we then can derive with the program structure

(20) $\tau \rightarrow F$.

Case $\tau \rightarrow \triangle\beta_{14}^j$

By the case assumption, we know

(21) $\tau \rightarrow \texttt{at}\{\beta_{15}^j\}$.

By the induction hypothesis (7) and Axiom 14 (While Loop), we know

(22) $\tau \rightarrow ((F \wedge P_j \neq \emptyset)\ \texttt{atprev}\ \beta_3^j)$.

By Lemma 1 (Locality) and Lemma 5 (Result Exclusion), we can derive

(23) $\tau \rightarrow G$.

Case $\tau \rightarrow \triangle(\texttt{at}\{\beta_{15}^j\} \wedge \neg\beta_{15}^j)$

By the case assumption, we know

(24) $\tau \rightarrow \texttt{at}\{\beta_{15}^j\}$.

By the induction hypothesis (7), we know

(25) $\tau \rightarrow \triangle G$.

By Lemma 1 (Locality) and Lemma 5 (Result Exclusion), we then know (8).

Case $\tau \rightarrow \triangle(\beta_{15}^j \wedge Q_j = \emptyset)$

By the case assumption, we know

(26) $\tau \rightarrow \texttt{at}\{\beta_3^j\}$.

By the program structure and Lemma 1 (Locality), we know

(27) $\tau \rightarrow found_j$.

By the induction hypothesis (7), we know

(28) $\tau \rightarrow \triangle G$.

Lemma 5 (Result Exclusion) therefore implies

(29) $\tau \rightarrow E$.

Case $\tau \rightarrow \triangle\beta^j_{20}$

By the case assumption, we know

(30) $\tau \rightarrow \texttt{at}\{\beta^j_{15}\}$.

By the induction hypothesis (7) and Axiom 14 (While Loop), we know

(31) $\tau \rightarrow ((G \wedge Q_j \neq \emptyset)\ \texttt{atprev}\ \beta^j_5)$.

By Lemma 6 (Set Update) and Lemma 5 (Result Exclusion), we can then derive

(32) $\tau \rightarrow G$. □

The following lemma describes the state of Π when the outer loop is left i.e. immediately before the critical region is left.

Lemma 26 (Wait End) *At the end of the main loop of $wait_j$, the process j has either found a result or it is going to be blocked:*

$$\begin{array}{l} \tau_e \rightarrow \\ \quad \neg run_j \wedge \\ \quad (found_j \rightarrow (\forall i(\neg \texttt{elem}_i(j, W_i)) \wedge r_j \in \overline{R}_j \wedge y_r = \overline{x}_{r_j})) \wedge \\ \quad (\neg found_j \rightarrow (\forall i \in \overline{R}_j(\texttt{elem}_i(j, W_i)) \wedge \forall i \notin \overline{R}_j(\neg \texttt{elem}_i(j, W_i))))). \end{array}$$

Proof: This is a corollary of Lemma 23 (Not Run), Lemma 25 (Wait Loop) and Lemma 5 (Result Exclusion). □

The following two lemmas describe the invariants of the nested loops of $result_i$ and are simultaneously proved similar to the previous invariants. Of particular interest is the invariant of the outer loop which not only describes the state at the entry of the loop but more generally the state of Π when the critical region protected by $rsem_j$ is entered (which happens at the end of each iteration of the outer loop). Therefore we have to check all exits of this critical region in any process.

Lemma 27 (Erase Loop) *At the start of each iteration of the inner loop of $result_i$, w_i is exactly in the wait sets $W_{i'}$ of all $i' \in S_i$:*

$$\begin{array}{l} \tau_w \rightarrow \forall i(((\texttt{at}\{\alpha^i_{13}\} \wedge w_i = j) \rightarrow \\ \quad (\forall i' \in S_i(\texttt{elem}_{i'}(w_i, W_{i'})) \wedge \forall i' \in R_{w_i} \backslash S_i(\neg \texttt{elem}_{i'}(w_i, W_{i'})))) \ \texttt{until}\ \tau_r). \end{array}$$

Proof: The proof is contained in the proof of Lemma 28 (Result Loop). □

Lemma 28 (Result Loop) *If process j is a member of some wait set W_i and no process is in the critical region protected by $\overline{rsem_j}$, then j is actually a member of all wait sets W_i with $i \in \overline{R}_j$ and j is (at the point of being) blocked:*

$$\begin{array}{l} \tau_w \rightarrow (((rsem_j \wedge \exists i(\texttt{elem}_i(j, W_i))) \rightarrow \\ \quad (\neg found_j \wedge \neg run_j \wedge \forall i \in \overline{R}_j(\texttt{elem}_i(j, W_i)) \wedge \texttt{at}\{\beta^j_{22}, \beta^j_{24}\}))\ \texttt{until}\ \tau_r). \end{array}$$

Proof: We prove this lemma simultaneously with Lemma 27 (Erase Loop). Let i be arbitrary fixed and define

(1) $X_0 \equiv rsem_j \wedge \exists i'(\mathtt{elem}_{i'}(j, W_{i'}))$,

(2) $X_1 \equiv \neg found_j \wedge \neg run_j \wedge \forall i' \in \overline{R}_j(\mathtt{elem}_{i'}(j, W_{i'})) \wedge \mathtt{at}\{\beta^j_{21}, \beta^j_{22}\}$,

(3) $Y_0 \equiv \mathtt{at}\{\alpha^i_{13}\} \wedge w_i = j$,

(4) $Y_1 \equiv \forall i' \in S_i(\mathtt{elem}_{i'}(w_i, W_{i'})) \wedge \forall i' \in R_{w_i} \backslash S_i(\neg\mathtt{elem}_{i'}(w_i, W_{i'}))$.

We want to prove

(5) $\tau_w \rightarrow ((X_0 \rightarrow X_1) \wedge (Y_0 \rightarrow Y_1))$ $\mathtt{until}$ τ_r.

We know by Lemma 19 (Entry)

(6) $\tau_w \rightarrow (\neg X_0 \wedge \neg Y_0)$

Let time τ be arbitrary fixed with $\tau_w < \tau < \tau_r$ and assume

(7) $\tau_w \rightarrow (((X_0 \rightarrow X_1) \wedge (Y_0 \rightarrow Y_1))$ $\mathtt{until}$ $\tau)$.

We have to prove

(8) $\tau \rightarrow ((X_0 \rightarrow X_1) \wedge (Y_0 \rightarrow Y_1))$.

We may restrict our considerations to the following cases:

1. $\tau \rightarrow \triangle\beta^{\mathbf{j}}_{\mathbf{21}}$: the critical region protected by $rsem_j$ is left and X_0 may become true.
2. $\tau \rightarrow \triangle(\beta^{\mathbf{j}}_{\mathbf{22}} \vee \beta^{\mathbf{j}}_{\mathbf{24}})$: $\mathtt{at}\{\beta^j_{22}, \beta^j_{24}\}$ may become false.
3. $\tau \rightarrow \triangle(\mathbf{w_i} = \mathbf{j} \wedge (\alpha^{\mathbf{i}}_{\mathbf{12}} \vee \mathtt{at}\{\alpha^{\mathbf{i}}_{\mathbf{13}}\} \vee \alpha^{\mathbf{i}}_{\mathbf{18}}))$: Y_0 becomes true.
4. $\tau \rightarrow \triangle(\alpha^{\mathbf{i'}}_{\mathbf{22}} \wedge \mathbf{w_{i'}} = \mathbf{j})$: the critical region protected by $rsem_j$ is left and X_0 may become true.

For all other actions, obviously either

- $\tau \rightarrow \neg X_0 \wedge \neg Y_0$ i.e. the premises are false or
- $\tau \rightarrow (\triangle X_1 \rightarrow X_1)$ and $\tau \rightarrow (\triangle Y_1 \rightarrow Y_1)$ i.e. the conclusions are not changed.

Case $\tau \rightarrow \triangle\beta^{\mathbf{j}}_{\mathbf{21}}$

We assume

(9) $\tau \rightarrow X_0$.

We then know by the case assumption and Lemma 26 (Wait End)

(10) $\tau \rightarrow \triangle(\neg run_j \wedge found_j \wedge \forall i \in \overline{R}_j(\mathtt{elem}_i(j, W_i)))$

and thus by the program structure also

(11) $\tau \rightarrow (\neg run_j \wedge found_j \wedge \forall i \in \overline{R}_j(\mathtt{elem}_i(j, W_i)) \wedge \mathtt{at}\{\beta^j_{22}\})$.

Therefore, we know

(12) $\tau \rightarrow X_1$.

By Lemma 5 (Result Exclusion), we also know

(13) $\tau \rightarrow \neg Y_0$.

Case $\tau \rightarrow \triangle\beta^{\mathbf{j}}_{\mathbf{22}}$

We assume

(14) $\tau \to X_0$.

The case assumption then implies

(15) $\tau \to \triangle X_0$

and thus by the induction hypothesis (7)

(16) $\tau \to \triangle X_1$.

This implies

(17) $\tau \to \triangle\neg \mathit{found}_j$

and thus by Axiom 13 (One Branch Conditional)

(18) $\tau \to \mathtt{at}\{\beta^j_{24}\}$.

From this and (16), we know

(19) $\tau \to X_1$.

If we assume

(20) $\tau \to Y_0$

we also know by the program structure

(21) $\tau \to \triangle Y_0$

and thus by the induction hypothesis (7) and the program structure

(22) $\tau \to Y_1$.

Case $\tau \to \triangle\beta^{\mathbf{j}}_{\mathbf{24}}$

The case assumption implies by Axiom 11 (Wait Statement)

(23) $\tau \to \triangle run_j$.

Then we know by the induction hypothesis (7)

(24) $\tau \to \triangle\neg X_0$

and by the program structure also

(25) $\tau \to \neg X_0$.

Let us assume

(26) $\tau \to Y_0$.

Then we know by the case assumption

(27) $\tau \to \triangle Y_0$

and thus by the induction hypothesis (7) and the program structure

(28) $\tau \to Y_1$.

Case $\tau \to \triangle(\alpha^{\mathbf{i}}_{\mathbf{12}} \wedge \mathbf{w_i} = \mathbf{j})$

By Axiom 11 (Wait Statement) and Lemma 22 (Lock), we know

(29) $\tau \to ((rsem_j \wedge \mathtt{elem}_i(j, W_i))\ \mathtt{atprev}\ \alpha^i_7)$.

By the induction hypothesis (7), we thus know

(30) $\tau \to (X_0\ \mathtt{atprev}\ \alpha^i_7)$.

By the program structure, Lemma 5 (Result Exclusion) and Lemma 2 (Wait Semaphore), we then know

(31) $\tau \rightarrow \forall i' \in \overline{R}_j(\mathtt{elem}_{i'}(j, W_{i'}))$

and thus also

(32) $\tau \rightarrow Y_1$.

By Lemma 5 (Result Exclusion), we also know

(33) $\tau \rightarrow \neg X_0$.

Case $\boldsymbol{\tau \rightarrow \triangle(\mathtt{at}\{\alpha^{i}_{13}\} \wedge w_i = j)}$

By the induction hypothesis (7) we have

(34) $\tau \rightarrow \triangle Y_1$

and thus by Lemma 5 (Result Exclusion)

(35) $\tau \rightarrow Y_1$.

By Lemma 5 (Result Exclusion), we also know

(36) $\tau \rightarrow \neg X$.

Case $\boldsymbol{\tau \rightarrow \triangle(\alpha^{i}_{18} \wedge w_i = j)}$

By the induction hypothesis (7) and Axiom 14 (While Loop), we know

(37) $\tau \rightarrow ((Y_1 \wedge S_i \neq \emptyset)\ \mathtt{atprev}\ \alpha^{i}_{13})$.

ant then by Lemma 5 (Result Exclusion) and Lemma 2 (Wait Semaphore)

(38) $\tau \rightarrow (\forall i' \in S_i(\mathtt{elem}_{i'}(j, W_{i'})) \wedge \forall i' \in \overline{R}_j \backslash S_i(\neg\mathtt{elem}_{i'}(j, W_{i'})))$.

By Lemma 5 (Result Exclusion), we then have

(39) $\tau \rightarrow Y_1$.

By Lemma 5 (Result Exclusion), we also know

(40) $\tau \rightarrow \neg X_0$.

Case $\boldsymbol{\tau \rightarrow \triangle(\alpha^{i'}_{23} \wedge w_{i'} = j)}$

We know by Lemma 5 (Result Exclusion)

(41) $\tau \rightarrow \neg Y_0$.

We know by Axiom 11 (Wait Statement)

(42) $\tau \rightarrow (rsem_j\ \mathtt{atprev}\ \alpha^{i'}_7))$.

We will now distinguish two cases:

1. **Case $\boldsymbol{\tau \rightarrow (j \in W_{i'}\ \mathtt{atprev}\ \alpha^{i'}_7)}$**

   We will show

   (43) $\tau \rightarrow \neg X_0$

   by proving

   (44) $\tau \rightarrow \forall k(\neg\mathtt{elem}_k(j, W_k))$.

   By the induction hypothesis (7), we know

   (45) $\tau \rightarrow ((\mathtt{elem}_{i'}(j, W_{i'}) \wedge \forall k \in R_j(\mathtt{elem}_k(j, W_k)))\ \mathtt{atprev}\ \alpha^{i'}_7)$.

Then we also have, by Lemma 5 (Result Exclusion),

(46) $\tau \to (j \in W_{i'}$ `atprev` $\alpha_9^{i'})$.

From this and the program structure, we then know

(47) $\tau \to ((\bigcirc\,\mathtt{at}\{\alpha_{12}^{i'}\})$ `atprev` $\alpha_{11}^{i'})$.

Furthermore, by the induction hypothesis (7) we know

(48) $\tau \to (\forall k(\neg\mathtt{elem}_k(j, W_k))$ `atprev` $\alpha_{19}^{i'})$

which leads by Lemma 5 (Result Exclusion) to (44).

2. **Case $\boldsymbol{\tau \to (j \notin W_{i'}}$ `atprev` $\boldsymbol{\alpha_7^{i'})}$**

   We assume

   (49) $\tau \to X_0$.

   By Lemma 5 (Result Exclusion), we know

   (50) $\tau \to (w_{i'} \notin W_{i'}$ `atprev` $\alpha_9^{i'})$

   and thus by Axiom 13 (One Branch Conditional)

   (51) $\tau \to ((\bigcirc\,\mathtt{at}\{\alpha_{22}^{i'}\})$ `atprev` $\alpha_{11}^{i'})$

   From this we know by (49) and Lemma 5 (Result Exclusion)

   (52) $\tau \to (X_0$ `atprev` $\alpha_7^{i'})$.

   From the induction hypothesis (7), we thus know

   (53) $\tau \to (X_1$ `atprev` $\alpha_7^{i'})$

   Then we have by (50), (51) and Lemma 5 (Result Exclusion)

   (54) $\tau \to X_1$. $\square$

The final case distinction in the previous proof actually describes an interesting situation. It may happen that some Π_i with $i \in R_j$ finds at α_5 that $j \in W_i$ but has to detect at α_9 that $j \notin W_i$! In this case, some other $\Pi_{i'}$ with $i' \in R_j$ has executed α_5 simultaneously to Π_i but was slightly faster in executing α_7. It "stole" therefore j from W_i and delivered $x_{i'}$ to j. The lock of $rsem_j$ at α_7 is therefore necessary to determine a unique winner of the "race".

Lemma 29 (Result End) *If process i is going to awake process j, j is not in any wait set $W_{i'}$ any more:*

$$\tau_w \to \Box\forall i((\alpha_{21}^i \wedge w_i = j) \to \forall i'(\neg\mathtt{elem}_{i'}(j, W_{i'})))$$

Proof: Let i be arbitrary fixed and let $\tau > \tau_w$ such that

(1) $\tau \to (\alpha_{21}^i \wedge w_i = j)$.

By the program structure, we then know

(2) $\tau \to (S_i = \emptyset$ `atprev` $\alpha_{13}^i)$.

By Lemma 27 (Erase Loop), this implies

(3) $\tau \to (\forall i'(\neg\mathtt{elem}_{i'}(j, W_{i'})$ `atprev` $\alpha_{13}^i)$.

By Lemma 5 (Result Exclusion), we therefore know

(4) $\tau \to \forall i'(\neg\mathtt{elem}_{i'}(j, W_{i'}))$. $\square$

The last lemma in this subsection describes the state of Π immediately before $wait_j$ is left. This lemma is necessary for two purposes:

1. To guarantee that Lemma 19 (Entry) holds for the next execution of $wait_j$.
2. To guarantee that, after some Π_i has set the values of r_j and y_j, they are not changed any more by any $\Pi_{i'}$.

Lemma 30 (Exit) *When $wait_j$ is left, $found_j$ and run_j are false and j is not in any wait set W_i and no process i is going to deliver its result to j:*

$$t_f \to ((\neg found_j \land \neg run_j \land \forall i(\neg \mathtt{elem}_i(j, W_i) \land (\mathtt{at}\{\alpha_1 2^i \ldots \alpha_{21}^i\} \to w_i \neq j))) \\ \mathtt{until}\ \tau_r)$$

Proof: By Lemma 23 (Not Run), we know

(1) $\tau_e \to \neg run_j$.

We will now distinguish two cases.

1. **Case $\tau_e \to$ found$_j$**

   By the program structure and Lemma 1 (Locality), we know

   (2) $\tau_e \to ((\neg found_j)\ \mathtt{until}\ \tau_r)$.

   By Lemma 4 (Result Semaphore), we know

   (3) $\tau_e \to \forall i(\mathtt{at}\{\alpha_8^i \ldots \alpha_{22}^i\} \to w_i \neq j)$.

   By (1), Lemma 26 (Wait End) and the program structure, we know

   (4) $\tau_e \to (\forall i(\neg \mathtt{elem}_i(j, W_i))\ \mathtt{until}\ \tau_r)$

   and with (3) therefore

   (5) $\tau_e \to (\forall i((\alpha_9^i \land w_i = j) \to (w_i \notin W_j))\ \mathtt{until}\ \tau_r)$

   By (1) and the program structure, it is then easy to derive

   (6) $\tau_e \to (B\ \mathtt{upto}\ \tau_r)$.

2. **Case $\tau_e \to \neg$found$_j$**

   By the program structure, we know

   (7) $\tau_e \to (\beta_{24}^j\ \mathtt{before}\ \tau_r)$.

   Define τ_g with $\tau_e < \tau_g < \tau_r$ such that

   (8) $\tau_e \to (\tau_g\ \mathtt{atnext}\ \beta_{24}^j)$.

   By Axiom 11 (Wait Statement), we know

   (9) $\tau_g \to run_j$.

   By (1) this implies for some i

   (10) $\tau_e \to ((\alpha_{21}^i \land w_i = j)\ \mathtt{before}\ \tau_g)$.

   Therefore, we know by the program structure and Lemma 29 (Result End)

   (11) $\tau_e \to (\forall i(\neg \mathtt{elem}_i(j, W_i))\ \mathtt{before}\ \tau_g)$.

   From this and the definition of α_{21}^i, we can derive

   (12) $\tau_f \to (B\ \mathtt{upto}\ \tau_r)$

   and are done. □

B.5 Continuation Lemmas

In the section, we deal with all lemmas expressing eventuality properties. In our case, these are assertions that

1. Critical regions protected by a semaphore are left.
2. While loops terminate.

The organization of proofs is as follows:

1. **Wait Continuation:** Lemma 31 says that the critical regions protected by $wsem_i$ are left. These regions do not contain other critical regions or while loops.
2. **Remove Induction:** Lemma 32 says that the inner loop of $wait_j$ (which only contains $wsem_i$ regions) terminates.
3. **Wait Induction:** Lemma 33 says that the outer loop of $wait_j$ (which contains the $wsem_i$ region and an inner loop) terminates.
4. **Erase Induction:** Lemma 34 says that the inner loop of $result_i$ (which contains a critical region protected by $wsem_i$) terminates.
5. **Result Continuation:** Lemma 35 says that the critical regions protected by $rsem_j$ are left. These regions contain the outer loop of $wait_j$ and the inner loop of $result_i$.
6. **Result Induction:** Lemma 38 shows that the outer loop of $result_i$ (which contains critical regions protected by $rsem_j$ and $wsem_i$ and an inner loop) terminates.
7. **First Out:** Lemma 39 provides an alternative condition that is sufficient to prove Theorem 5 (Sometimes).

Lemma 31 (Wait Continuation) *No critical region protected by some $wsem_i$ will be occupied forever:*

$$\tau_\Pi \to \Box\forall i(\Diamond wsem_i).$$

Proof: Let time $\tau \geq \tau_\Pi$ and i be arbitrary fixed and assume

(1) $\tau \to \neg wsem_i$.

By Lemma 2 (Wait Semaphore), we then know

(2) $\tau \to \mathtt{at}\{\alpha^i_4 \ldots \alpha^i_6, \alpha^i_9, \alpha^i_{10}, \alpha^i_{24}\} \vee$
(3) $\tau \to \exists i'(s_{i'} = i \wedge \mathtt{at}\{\alpha^{i'}_{16}, \alpha^{i'}_{17}\}) \vee$
(4) $\tau \to \exists j(p_j = i \wedge \mathtt{at}\{\beta^j_6 \ldots \beta^j_8, \beta^j_{10}\}) \vee$
(5) $\tau \to \exists j(q_j = i \wedge \mathtt{at}\{\beta^j_{18}, \beta^j_{19}\})$.

All of the denoted regions are terminated by $\mathbf{V}(wsem_i)$ and no one contains an **endwhile** or a **P** statement. Hence, we know by Axiom 16 (Fairness)

(6) $\tau \to \Diamond wsem_i$. □

Lemma 32 (Remove Induction) *The inner loop in $wait_j$ terminates:*

$$\tau_\Pi \to \Box(\beta^j_{15} \to \Diamond(\beta^j_{15} \wedge Q_j = \emptyset)).$$

Proof: This can be easily verified by induction over Q_j i.e. by proving the assertion

$$\begin{aligned}&\tau_\Pi \to \Box\forall\overline{Q}(\\ &\quad(\beta^j_{15} \wedge Q_j = \overline{Q} \wedge \overline{Q} \neq \emptyset) \to\\ &\qquad((\bigcirc\Diamond\beta^j_{15}) \wedge (Q_j \subset \overline{Q}\ \mathtt{atnext}\ \beta^j_{15})))\end{aligned}$$

by the application of Lemma 31 (Wait Continuation) and Axiom 16 (Fairness). □

Lemma 33 (Wait Induction) *The outer loop in $wait_j$ terminates:*

$$\tau_\Pi \to \Box(\beta_3^j \to \Diamond(\beta_3^j \wedge (P_j = \emptyset \vee found_j)).$$

Proof: We will prove this by induction over P_j i.e. by proving the assertion

$$\begin{array}{l} \tau_\Pi \to \Box\forall\overline{P}(\\ \quad (\beta_3^j \wedge P_j = \overline{P} \wedge \overline{P} \neq \emptyset \wedge \neg found_j) \to \\ \qquad ((\bigcirc\Diamond\beta_3^j) \wedge ((found_j \vee P_j \subset \overline{P}) \ \mathtt{atnext}\ \beta_3^j))). \end{array}$$

Let time $\tau > \tau_\Pi$ and $\overline{P}$ be arbitrary fixed such that

(1) $\tau \to (\beta_3^j \wedge P_j = \overline{P} \wedge \overline{P} \neq \emptyset)$.

By Axiom 16 (Fairness), it is clear that

(2) $\tau \to \bigcirc\Diamond\beta_6^j$.

We will distinguish two cases:

1. Case $\boldsymbol{\tau \to (\neg \mathbf{avail}_{P_j}\ \mathtt{atnext}\ \beta_6^j)}$

   By Lemma 31 (Wait Continuation) and Lemma 16 (Fairness), we know

   (3) $\tau \to \bigcirc\Diamond\beta_3^j$.

   Furthermore, we can easily show

   (4) $\tau \to (P_j \subset \overline{P}\ \mathtt{atnext}\ \beta_3^j)$.

2. Case $\boldsymbol{\tau \to (\mathbf{avail}_{P_j}\ \mathtt{atnext}\ \beta_5^j)}$

   By Lemma 32 (Remove Induction), we can prove

   (5) $\tau \to \bigcirc\Diamond\beta_3^j$.

   Furthermore, by the program structure it is easy to show

   (6) $\tau \to (found_j\ \mathtt{atnext}\ \beta_3^j)$. □

Lemma 34 (Erase Induction) *The inner loop in $result_i$ terminates:*

$$\tau_\Pi \to \Box\forall i(\alpha_{13}^i \to \Diamond(\alpha_{13}^i \wedge S_i = \emptyset)).$$

Proof: This can be easily verified by induction over S_i i.e. by proving the assertion

$$\begin{array}{l} \tau_\Pi \to \Box\forall i\forall\overline{S}(\\ \quad (\alpha_{16}^i \wedge S_i = \overline{S} \wedge \overline{S} \neq \emptyset) \to \\ \qquad ((\bigcirc\Diamond\alpha_{16}^i) \wedge (S_i \subset \overline{S}\ \mathtt{atnext}\ \alpha_{16}^i))) \end{array}$$

by the application of Lemma 31 (Wait Continuation) and Axiom 16 (Fairness). □

Lemma 35 (Result Continuation) *No critical region protected by some $rsem_j$ will be occupied forever:*

$$\tau_\Pi \to \Box\Diamond rsem_j.$$

Proof: Let time $\tau \geq \tau_\Pi$ be arbitrary fixed such that

(1) $\tau \to \neg rsem_j$.

We will prove

(2) $\tau \to \Diamond rsem_j$.

By Lemma 4 (Result Semaphore), we have one the following cases:

1. **Case $\tau \rightarrow$ at$\{\beta_3^j \dots \beta_{21}^j\}$**

   By Lemma 33 (Wait Induction), we can prove

   (3) $\tau \rightarrow \Diamond(\beta_3^j \wedge (P_j = \emptyset \vee found_j))$.

   By Axiom 16 (Fairness), we then know

   (4) $\tau \rightarrow \Diamond\beta_{20}^j$

   and are done.

2. **Case $\tau \rightarrow (\mathbf{w_i} = \mathbf{j} \wedge$ at$\{\alpha_8^i \dots \alpha_{22}^i\})$**

   By Lemma 31 (Wait Continuation), we know

   (5) $\tau \rightarrow \Diamond\alpha_9^i$.

   We may assume

   (6) $\tau \rightarrow (j \in W_i$ `atnext` $\alpha_9^i)$

   since otherwise Axiom 13 (One Branch Conditional) and Axiom 16 (Fairness) would directly lead to (8). Then we know by the program structure and Lemma 34 (Erase Induction)

   (7) $\tau \rightarrow \Diamond(\alpha_{13}^i \wedge S_i = \emptyset)$

   which leads to

   (8) $\tau \rightarrow \Diamond\alpha_{22}^i$. □

The following lemma says that $result_j$ puts j only in such W_i where the corresponding Π_i has not yet terminated.

Lemma 36 (Terminate) *If process j has put the identifier j into some wait set W_i, then the process i has not yet terminated:*

$$\tau_w \rightarrow ((\texttt{at}\{\beta_3^j\} \wedge \neg found_j) \rightarrow \forall i \in \overline{R}_j \backslash P_j(\neg\,\texttt{at}\{\alpha_{24}^i, \alpha_e^i\}))\ \texttt{until}\ \tau_r.$$

Proof: Define

(1) $D \equiv \forall i \in \overline{R}_j \backslash P_j(\neg\,\texttt{at}\{\alpha_{24}^j, \alpha_e^j\}))$.

By the definition of τ_w, we know

(2) $\tau_w \rightarrow \neg\,\texttt{at}\{\beta_3^j\}$.

Let time τ be arbitrary fixed with $\tau_w < \tau < \tau_r$ and assume

(3) $\tau \rightarrow (\texttt{at}\{\beta_3^j\} \wedge \neg found_j)$.

We are going to prove

(4) $\tau \rightarrow D$.

We know by the program structure

(5) $\tau \rightarrow \triangle((\texttt{at}\{\beta_3^j\} \wedge \neg\beta_3^j) \vee \beta_2^j \vee \beta_9^j \vee \beta_{15}^j)$

and may therefore restrict our considerations to the following cases.

Case $\tau \rightarrow \triangle(at\{\beta_3^j\} \wedge \neg\beta_3^j)$

By the induction hypothesis, we know

(6) $\tau \rightarrow \triangle\forall i \in \overline{R}_j \backslash P_j(\neg\,\texttt{at}\{\alpha_{23}^i, \alpha_e^i\})$.

By (3) and Lemma 25 (Wait Loop), we also know

(7) $\tau \rightarrow \triangle\forall i \in \overline{R}_j \backslash P_j(\texttt{elem}_i(j, P_i))$.

and therefore

(8) $\tau \to \triangle \forall i \in \overline{R}_j(\alpha_4^i \to W_i \neq \emptyset)$

and are thus done.

Case $\tau \to \triangle\beta_2^j$

By the program structure and Lemma 1 (Locality), we know

(9) $\tau \to P_j = \overline{R}_j$

and are therefore done.

Case $\tau \to \triangle\beta_{15}^j$

By the program structure and Lemma 14 (Found), we know

(10) $\tau \to \mathit{found}_j$

and are therefore done.

Case $\tau \to \triangle\beta_9^j$

Let i be arbitrary fixed such that

(11) $\tau \to i \in \overline{R}_j \backslash P_j$.

Let time $\tau_1 > \tau_w$ such that

(12) $\tau \to (\tau_1 \texttt{ atprev } \beta_3^j)$.

We will distinguish two cases:

1. **Case $\tau_1 \to i \in \overline{R}_j \backslash P_j$**

   By the induction hypothesis, we know

   (13) $\tau_1 \to \neg\, \texttt{at}\{\alpha_{24}^i, \alpha_e^i\}$.

   Lemma 25 (Wait Loop) and (3) imply

   (14) $\tau_1 \to \texttt{elem}_i(j, W_i)$

   By Lemma 4 (Result Semaphore), we also know

   (15) $\tau_1 \to (\neg \mathit{rsem}_j \texttt{ until } \tau)$

   and thus by Lemma 5 (Result Exclusion)

   (16) $\tau_1 \to (\texttt{elem}_i(j, W_i) \texttt{ until } \tau)$.

   But then we also know

   (17) $\tau_1 \to ((\alpha_4^i \to W_i \neq \emptyset) \texttt{ until } \tau)$

   and are therefore done.

2. **Case $\tau_1 \to i \in P_j$**

   Let time τ_2 such that

   (18) $\tau \to (\tau_2 \texttt{ atprev } \beta_6^j)$.

   We know by the case assumption and Axiom 12 (Two Branch Conditional)

   (19) $\tau_2 \to (p_j = i \wedge \neg \mathit{avail}_i)$

   and therefore by Lemma 9 (Available)

   (20) $\tau_2 \to \neg\, \texttt{at}\{\alpha_3^i \ldots \alpha_e^i\}$.

Let τ_3 such that

(21) $\tau_2 \rightarrow (\tau_3 \texttt{ atnext } \beta_8^j)$.

From Lemma 2 (Wait Semaphore) and (20), we can conclude

(22) $\tau_3 \rightarrow \neg\, \texttt{at}\{\alpha_4^i \ldots \alpha_e^i\}$.

Furthermore, we know by the program structure and Lemma 5 (Result Exclusion)

(23) $\tau_3 \rightarrow \texttt{elem}_i(j, W_i)$.

From both assertions we can conclude

(24) $\tau_3 \rightarrow ((\alpha_4^i \rightarrow j \in W_i) \texttt{ until } \tau)$

and therefore

(25) $\tau_3 \rightarrow (\neg\, \texttt{at}\{\alpha_{24}^i, \alpha_e^i\} \texttt{ until } \tau)$. □

Lemma 37 (Iteration) *If j is in W_i outside the critical region protected by $rsem_j$, then eventually a new iteration of the outer loop of $result_j$ will start.*

$$\tau_w \rightarrow (\forall i((rsem_j \wedge \texttt{elem}_i(j, W_i)) \rightarrow \Diamond\alpha_4^i \texttt{ until } \tau_r).$$

Proof: Let τ with $\tau_r \leq \tau < \tau_w$ and i be arbitrary fixed such that

(1) $\tau \rightarrow rsem_j \wedge \texttt{elem}_i(j, W_i)$.

By Lemma 21 (Waiting), we know

(2) $\tau_w \rightarrow (\beta_{21}^j \texttt{ before } \tau)$.

By the program structure, there exists a time τ' with $\tau_w < \tau' < \tau$ such that

(3) $\tau' \rightarrow (\beta_3^j \wedge \neg found_j \wedge P_j = \emptyset)$.

By Lemma 36 (Terminate) and Lemma 20 (Membership), we then know

(4) $\tau' \rightarrow \neg\, \texttt{at}\{\alpha_{24}^i, \alpha_e^i\}$

i.e. we know by Lemma 27 (Erase Loop) and Lemma 35 (Result Continuation)

(5) $\tau' \rightarrow \Diamond\alpha_4^i$

and are done. □

The following lemma shows that not only $wait_j$ but also $result_i$ terminates. This is the main constituent in the proof of Theorem 5 (Sometime).

Lemma 38 (Result Induction) *The outer loop in $result_i$ terminates:*

$$\tau_w \rightarrow \forall i((rsem_j \wedge \texttt{elem}_i(j, W_i)) \rightarrow \Diamond(\alpha_4^i \wedge W_i = \emptyset)) \texttt{ until } \tau_r.$$

Proof: Let τ with $\tau_w < \tau < \tau_r$ and i be arbitrary fixed such that

(1) $\tau \rightarrow (rsem_j \wedge \texttt{elem}_i(j, W_i))$.

By Lemma 37 (Iteration), we know

(2) $\tau \rightarrow \Diamond\alpha_4^i$.

Let us assume that for some $\overline{W} \neq \emptyset$

(3) $\tau' \rightarrow W_i = \overline{W}$.

It now suffices to show

(4) $\tau' \rightarrow (W_i \subset \overline{W} \texttt{ atnext } \alpha_4^i)$.

By Lemma 9 (Available), we know

(5) $\tau' \rightarrow \Box avail_i$.

This implies by Lemma 2 (Wait Semaphore)

(6) $\tau' \to \Box\forall j(\beta_7^j \to p_j \neq i)$

i.e. W_i will be not extended any more by any other process. Define j such that

(7) $\tau' \to (w_i = j \texttt{ atnext } \alpha_6^i)$.

We know by Lemma 2 (Wait Semaphore)

(8) $\tau' \to j \in W_i$.

In case of

(9) $\tau' \to (j \notin W_i \texttt{ atnext } \alpha_9^i)$

we are done. Otherwise, Lemma 29 (Result End) implies

(10) $\tau' \to (j \notin W_i \texttt{ atnext } \alpha_{21}^i)$

and we are done. □

The following lemma can be proved using the additional Axiom 18 (Queue) and may be used in an alternative proof of Theorem 5 (Sometime) in case that (by a modification in the program structure) Lemma 38 (Result Induction) does not hold:

Lemma 39 (First Out) *No j will remain in W_i forever:*

$$\tau_w \to \forall i((rsem_j \wedge \texttt{elem}_i(j, W_i)) \to \Diamond(\neg\texttt{elem}_i(j, W_i))) \texttt{ until } \tau_r.$$

Proof: Let τ with $\tau_w < \tau < \tau_r$ and i be arbitrary fixed such that

(1) $\tau \to (rsem_j \wedge \texttt{elem}_i(j, W_i))$.

By Lemma 37 (Iteration), we know that for some time $\tau' > \tau$

(2) $\tau' \to \alpha_4^i$.

We may assume

(3) $\tau' \to \texttt{elem}_i(j, W_i)$

since otherwise we are done. By Lemma 35 (Result Continuation) and Lemma 27 (Erase Loop), we know

(4) $\tau' \to \Box(\alpha_4^i \to (W_i = \emptyset \vee \bigcirc\Diamond\alpha_4^i))$

i.e. the iteration will continue forever unless W_i becomes empty. We may assume

(5) $\tau' \to \Box(\alpha_4^i \to \bigcirc\Diamond\alpha_4^i)$

since otherwise we are done (see Lemma 38 (Result Induction)). By the program structure and Lemma 27 (Erase Loop), we know for some j'

(6) $\tau' \to \bigcirc((j' = w_i \wedge w_i \notin W_i) \texttt{ atnext } \alpha_4^i)$.

We may assume $j \neq j'$ since otherwise we are done. But, by Axiom 18 (Queue), we know

(7) $\tau' \to j' \in W_i$.

Since there are only finitely many such j', we know with (5)

(8) $\tau' \to \Diamond w_i = j$

and are therefore done. □

C The Program Model

$\mathbf{result_i}(x_i) \equiv$

```
(α_1^i)     value_i := x_i
(α_2^i)     avail_i := true

(α_3^i)     P(wsem_i)
(α_4^i)     while W_i ≠ ∅ do
(α_5^i)         w_i ∈ W_i
(α_6^i)         V(wsem_i)

(α_7^i)         P(rsem_{w_i})

(α_8^i)         P(wsem_i)
(α_9^i)         deliver_i := w_i ∈ W_i
(α_10^i)        V(wsem_i)

(α_11^i)        if deliver_i then

(α_12^i)            S_i := R_{w_i}
(α_13^i)            while S_i ≠ ∅ do
(α_14^i)                s_i ∈ S_i
(α_15^i)                P(wsem_{s_i})
(α_16^i)                rem(W_{s_i}, w_i)
(α_17^i)                V(wsem_{s_i})
(α_18^i)                S_i := S_i \ {s_i}
                    endwhile

(α_19^i)            y_{w_i} := x_i
(α_20^i)            r_{w_i} := i
(α_21^i)            V(run_{w_i})

                endif

(α_22^i)        V(rsem_{w_i})

(α_23^i)        P(wsem_i)
            endwhile
(α_24^i)    V(wsem_i)

(α_e^i)     stop
```

$\textbf{wait}_{\mathbf{j}}(y_j, r_j, R_j) \equiv$

(β_1^j) $\quad P_j := R_j$

(β_2^j) $\quad \mathbf{P}(rsem_j)$
(β_3^j) $\quad$ **while** $P_j \neq \emptyset \wedge \neg found_j$ **do**

(β_4^j) $\qquad p_j \in P_j$
(β_5^j) $\qquad \mathbf{P}(wsem_{p_j})$

(β_6^j) $\qquad$ **if** $\neg avail_{p_j}$ **then**

(β_7^j) $\qquad\quad \mathbf{add}(W_{p_j}, j)$
(β_8^j) $\qquad\quad \mathbf{V}(wsem_{p_j})$
(β_9^j) $\qquad\quad P_j := P_j \backslash \{p_j\}$

$\qquad$ **else**
(β_{10}^j) $\qquad\quad \mathbf{V}(wsem_{p_j})$

(β_{11}^j) $\qquad\quad y_j := value_{p_j}$
(β_{12}^j) $\qquad\quad r_j := p_j$
(β_{13}^j) $\qquad\quad found_j := \mathrm{true}$

(β_{14}^j) $\qquad\quad Q_j := R_j \backslash P_j$
(β_{15}^j) $\qquad\quad$ **while** $Q_j \neq \emptyset$ **do**
(β_{16}^j) $\qquad\qquad q_j \in Q_j$
(β_{17}^j) $\qquad\qquad \mathbf{P}(wsem_{q_j})$
(β_{18}^j) $\qquad\qquad \mathbf{rem}(W_{q_j}, j)$
(β_{19}^j) $\qquad\qquad \mathbf{V}(wsem_{q_j})$
(β_{20}^j) $\qquad\qquad Q_j := Q_j \backslash \{q_j\}$
$\qquad\quad$ **endwhile**
$\qquad$ **endif**

$\quad$ **endwhile**
(β_{21}^j) $\quad \mathbf{V}(rsem_j)$

(β_{22}^j) $\quad$ **if** $found_j$ **then**
(β_{23}^j) $\qquad found_j := \mathrm{false}$
$\quad$ **else**
(β_{24}^j) $\qquad \mathbf{P}(run_j)$
$\quad$ **endif**

(β_e^j) $\quad$ **nop**