TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Formal derivations of non-blocking multiprograms

Citation for published version (APA):
Mooij, A. J. (2002). *Formal derivations of non-blocking multiprograms*. (Computer science reports; Vol. 0213). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/2002

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 23. Sep. 2021

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER'S THESIS

Formal derivations of non-blocking multiprograms

by

A.J. Mooij

Supervisor: ir. W.H.J. Feijen

Eindhoven, August 2002

Preface

This master's thesis results from my graduation project carried out under supervision of W.H.J. Feijen at the Department of Mathematics and Computer Science of Eindhoven University of Technology. The aim of this project was to make non-blocking multiprograms more accessible by showing how to derive them using the method of multiprogramming described in [FvG99].

In the last two decades the interest in non-blocking multiprograms has increased, resulting in a lot of publications. Most publications present some new non-blocking multiprograms with at best an operational description and some a-posteriori correctness proof. Unfortunately, they provide hardly any insight in the *construction* of non-blocking multiprograms. In this thesis, however, we focus on the construction (and on the fly the correctness proof) of such multiprograms. We did not intend to design new multiprograms.

The body of this thesis consists of two main parts. The first part deals with methodological issues: it deals with our method of multiprogramming and with definitions of non-blockingness. The second part contains a number of derivations of non-blocking multiprograms.

A.J. Mooij
Helmond, The Netherlands
3 July 2002

Table of contents

0. Introduction to (non-blocking) synchronisation

0.0 Introduction

Whether we realise it or not, parallelism and synchronisation are part of everybody's daily life. They occur at all levels in nature, for example groups of humans that communicate, and machines in a production line that co-operate.

A parallel system consists of a collection of components. The components operate concurrently and fairly autonomously. There are two major reasons for the interest in parallel systems in comparison to systems consisting of only one component: performance and fault-tolerance. A collection of components can potentially do more work in less time than a single component can, thus leading to a better performance. If a component in a parallel system stops operating, the system as a whole does not have to stop operating because other components can potentially continue, thus leading to a better fault-tolerance.

Usually the components cannot operate completely autonomously because they have to interact with other components (for example, to communicate). Such an interaction between components is called synchronisation. There are many ways of synchronisation, which are classified into two main classes: lock-based synchronisation and non-blocking synchronisation. These classes will be defined later on.

This chapter explains why one should study other classes of synchronisation than the traditional lock-based synchronisation. The aim of this chapter is not to provide a thorough introduction to parallelism and synchronisation, but to provide an overview of synchronisation and to motivate interest in non-blocking synchronisation.

The traditional lock-based synchronisation will be analysed in section 0.1. In section 0.2, non-blocking synchronisation will be analysed. In section 0.3, a subclass of non-blocking synchronisation, called wait-free synchronisation, will be introduced. Finally in section 0.4, the different classes of synchronisation will be compared to find out which class of synchronisation is best suitable to solve synchronisation problems. These subsections are intended to be read consecutively because in many subsections new terminology is defined, which is also used in later subsections.

0.1 Lock-based synchronisation

The synchronisation method that has received most attention so far is the so-called lock-based synchronisation. This way of synchronisation is called lock-based because a component that has to synchronise with some other components may become blocked. A blocked component is inactively waiting for some other components. So blocking is a special form of waiting.

Blocked components reduce the two benefits of parallel systems as mentioned above: performance and fault-tolerance. A blocked component does not exploit the potential performance improvement because it is inactive. Blocked components endanger the potential fault-tolerance because progress of blocked components depends on other components. If some components stop operating, a blocked component may even never be able to continue.

To enable a component to become blocked, one usually needs special blocking primitive operations. Because support from an operating system is usually required for these primitive operations, they are considered high-level primitive operations. We will not elaborate on them here.

0.1.1 Examples

To illustrate lock-based synchronisation, we will explain two examples of synchronisation problems that can easily be solved using lock-based synchronisation. First we will explain the mutual exclusion problem; then we will show how we could exploit a solution to this problem to solve another synchronisation problem.

The mutual exclusion problem considers a collection of components; each component may contain sections (i.e. consecutive operations) that are marked as so-called critical sections. The mutual exclusion problem is to synchronise these components such that at any moment in time at most one component operates in a critical section. To solve this problem, components that have to enter a critical section must be able to wait if another component is in a critical section. We will not treat this example in more detail.

Solutions to the mutual exclusion problem can be exploited to solve other synchronisation problems, for example the readers-writers problem. The readers-writers problem considers a collection of components that share a (large) communication buffer. There are two types of components: readers that read from the buffer and writers that write into the buffer. The main problem is to synchronise the components such that writing into the buffer does not overlap with reading or some other writing. To solve this problem we could mark the reading and writing sections as critical sections. Then the readers-writers problem reduces to the mutual exclusion problem described above.

0.1.2 Disadvantages

Although lock-based synchronisation is often used, it has some disadvantages (that will be explained below) with respect to:

- Primitive operations
- Degree of parallelism
- Fault-tolerance
- Priority inversion

Primitive operations: Blocking primitive operations usually need support from an operating system. Thus these operations may not always be available.

Degree of parallelism: If a component has become blocked, it is inactive. So blocked components cannot exploit the potential performance improvement of parallelism. This is called a reduced degree of parallelism and it is mainly a disadvantage if the blocking is unnecessarily.

Fault-tolerance: If a component is waiting for other components, its progress totally depends on the other components. So, if some components stop operating, the progress of waiting components may not be guaranteed.

Priority-inversion: In some parallel systems some components have a higher priority than others, because their progress is considered more important than the progress of others. Then a waiting component may be waiting for lower priority components, which is called priority-inversion. This may be undesirable.

0.2 Non-blocking synchronisation

The synchronisation method that has not yet received sufficient attention is non-blocking synchronisation. It is called non-blocking because components will never become blocked. Thus it overcomes the disadvantages of lock-based synchronisation with respect to:

- Primitive operations
- Degree of parallelism

0.2.1 Example

To illustrate non-blocking synchronisation, we will explain a way to obtain a non-blocking solution to a synchronisation problem on the basis of a lock-based solution. Note that we only have to focus on the blocking operations of the components. If we ensure that a component that could become blocked can decide whether it may already continue, we only have to transform the (inactive) blocking into active waiting. So the blocking operations of a component can be eliminated by replacing them by some repeatedly checking whether that component may continue. Thus a non-blocking solution is obtained that uses a usually called busy form of waiting.

0.2.2 Disadvantages

Although non-blocking solutions can easily be obtained from lock-based solutions, and although non-blocking synchronisation overcomes two of the disadvantages of lock-based synchronisation, it potentially has some disadvantages with respect to:

- Fault-tolerance
- Priority inversion
- Amount of extra work

Fault-tolerance: Just like lock-based synchronisation, a component that stops operating may endanger the progress of other components.

Priority-inversion: Just like lock-based synchronisation, components may be waiting for lower priority components.

Amount of extra work: Because the components cannot become blocked, they remain active to repeatedly check whether they may continue. In general, components may often perform such a check while they cannot yet continue. This involves a potentially unlimited amount of extra work, which may be undesirable.

0.3 Wait-free synchronisation

The disadvantages of lock-based and general non-blocking synchronisation are all caused by the fact that components may have to wait. To overcome these disadvantages, there is a special subclass of non-blocking synchronisation that is called wait-free synchronisation. Using wait-free synchronisation, components are synchronised in such a way that they do not have to wait for other components.

0.3.1 Examples

Before we illustrate wait-free synchronisation, we must first realise that it is not possible to solve all synchronisation problems using wait-free synchronisation. For example, the mutual exclusion problem cannot be solved.

We now reconsider the readers-writers problem. Because in a wait-free solution a component (that has to read or write) may not wait, a strategy to obtain a wait-free solution would be to introduce multiple copies of the buffer. Then it can be ensured that writing never occurs in a copy where another component is reading or writing. However, the amount of extra work and extra space usage may be considerable.

0.3.2 Disadvantages

Although wait-free synchronisation potentially overcomes all disadvantages of lock-based and general non-blocking synchronisation, it has some disadvantages with respect to:

- Primitive operations
- Amount of extra work
- Amount of extra space
- Complexity

Primitive operations: Sometimes special primitive operations are needed for wait-free synchronisation. Because these operations do not involve blocking, they do not need support from an operating system and hence they are considered lower level operations than the blocking primitive operations. Nevertheless, these operations may not always be available.

Amount of extra work: Because components cannot become blocked, the total amount of extra work that must be performed to synchronise may be high, but limited.

Amount of extra space: In some cases much extra space (e.g. multiple copies of a buffer) is needed for wait-free synchronisation.

Complexity: Developers generally consider it more difficult to solve a synchronisation problem using wait-free synchronisation in comparison to using lock-based synchronisation. This is probably due to fact that the primitive operations are of a lower level and that the components may not become blocked and hence are always active. Furthermore, there are few formal methods to solve synchronisation problems using wait-free synchronisation.

0.4 Evaluation

As one might expect, the disadvantages of wait-free synchronisation are overcome by lock-based or non-blocking synchronisation. Non-blocking synchronisation does not use special primitive operations. Lock-based synchronisation does not involve much extra work or space and there are formal methods to solve synchronisation problems using lock-based synchronisation.

The following overview summarises the properties of the synchronisation methods:

	Lock-based	**Non-blocking**	**Wait-free**
Primitive operations	-	+	0
Degree of parallelism	-	+	+
Fault-tolerance	-	-	+
Priority inversion	-	-	+
Amount of extra work	+	-	0
Amount of extra space	+	+	-
Complexity	+	+	-

+ = positive 0 = moderate - = negative

Which class of synchronisation is best for a given synchronisation problem is a trade-off between those factors. For general synchronisation with small amounts of extra work and space, lock-based synchronisation is probably best suitable. If blocking primitive operations are not available, general non-blocking synchronisation may be a good alternative. To optimally benefit from the potential increase in performance and fault-tolerance of parallelism, wait-free synchronisation may be very interesting.

0.5 Conclusion

The class of lock-based synchronisation has got most attention and it is appropriate in many cases. On the other hand, there are situations in which no blocking primitive operations are available or an optimal benefit from parallelism is required. Then the other classes of synchronisation could be preferable.

Although there exist many formal methods to solve synchronisation problems using lock-based synchronisation, there are few formal methods yet to solve synchronisation problems using non-blocking synchronisation or even wait-free synchronisation. This thesis results from my graduation project, in which I studied how to solve synchronisation problems using non-blocking and especially wait-free synchronisation using existing formal methods to solve these problems using lock-based synchronisation.

Methodological Issues

1. Summary of our method of multiprogramming

This chapter summarises our method of multiprogramming, which is based on [FvG99]. We assume the reader to be familiar with the predicate calculus.

1.0 Our computational model

A multiprogram is a set of components, i.e. sequential programs. Execution of a single component results in a sequence of atomic actions prescribed by the state and the program text. Execution of a multiprogram results in a sequence of atomic actions that is an interleaving of the sequences generated by the individual components.

To be able to guarantee progress for *all* components, [FvG99] assumes that a "fair" interleaving is generated. We will not make such assumption, because we will consider a computational model in which (undetectable) halting failures may occur. If at some point in the interleaving, a non-terminated component will, within a finite number of steps of the rest of the system, not contribute a next action to the interleaving, such component is a halted component. In such model we can only guarantee progress of the non-halted components, of which, by definition of a halted component, a fair interleaving is guaranteed.

1.1 Annotation and program derivation

We describe the program text of a component using the Guarded Command Language. In between two subsequent atomic statements are so-called control-points.

We annotate our programs with assertions. An assertion is a predicate on the state space that is placed at a control point. An assertion at a control point is correct if the state satisfies that assertion whenever a component is at that control point. A queried assertion is an assertion that has not yet been proved to be correct. Usually an assertion Q is denoted as $\{Q\}$, a queried assertion Q as $\{?\ Q\}$, and a statement S with pre-assertion Q as $\{Q\}\ S$.

For correctness of the annotation we use the Owicki/Gries theory. According to that theory an assertion P in a component is correct whenever

- local correctness is guaranteed, i.e. if it is an initial assertion it is implied by the precondition of the multiprogram, and if it is preceded by atomic statement $\{Q\}\ S$ it is established by that statement, i.e. $\{Q\}\ S\ \{P\}$ is a correct Hoare-triple; and
- global correctness (sometimes called maintenance or interference freedom) under each atomic statement $\{Q\}\ S$ in the other components is guaranteed, i.e. $\{P \wedge Q\}\ S\ \{P\}$ is a correct Hoare-triple.

A system invariant is an assertion that can be added at each control point and hence, we have the freedom of not writing it anywhere in our annotation. Thus a system invariant P is correct whenever

- initial correctness is guaranteed, i.e. it is implied by the precondition of the multiprogram; and
- it is maintained by each atomic statement $\{Q\}\ S$ of the multiprogram, i.e. $\{P \wedge Q\}\ S\ \{P\}$ is a correct Hoare-triple.

Our strategy in program derivation is to repeatedly strengthen the annotation until all assertions are correct. The strengthening steps will be guided by the desire to turn a queried assertion into a correct one. Therefore we will deal with one queried assertion at a time.

To ensure that a queried assertion becomes correct we may have to strengthen the current annotation with fresh queried assertions. Fortunately, such strengthening of the annotation can not endanger the correctness of the current annotation. Sometimes we may have to insert a statement preceding that assertion. The places where we may insert statements are usually indicated by anonymous placeholders, to be denoted as "…". Upon insertion of a statement, maintenance of all assertions in the other components can be endangered.

1.2 Statements and proof rules

Operationally, a Hoare-triple {P} S {Q} is a boolean that has the value *true* if and only if each terminating execution of *S* that starts from an initial state satisfying *P* is guaranteed to end up in a final state satisfying *Q*. The weakest liberal precondition for statement *S* and predicate *Q*, to be denoted by *wlp.S.Q*, is the weakest precondition *P* such that {P} S {Q}. That is we have: $\{P\}\ S\ \{Q\} \equiv [\ P \Rightarrow \text{wlp.S.Q}\ ]$

In the remainder of this section we give the main atomicity and proof rules of each statement of the Guarded Command Language. Note that a non-atomic action *S* can be made atomic by inserting it within atomic brackets: < S >. We will start with the two elementary statements and then we will continue with the three possibilities for construction.

1.2.1 The skip

The **skip**, which does nothing, is an atomic action. By definition,

$[\ \text{wlp.}\mathbf{skip}\text{.Q} \equiv Q\ ]$.

In practice we rely on the following little theorem:

$\{P\}\ \mathbf{skip}\ \{Q\} \equiv [\ P \Rightarrow Q\ ]$.

1.2.2 The assignment

A simple assignment like *x, y := E, F* evaluates expressions *E* and *F* and then assigns their values to variables *x* and *y* respectively. A complex assignment like *x, y : E.x.y* non-deterministically assigns to variables *x* and *y* values *x'* and *y'* respectively for which *E.x'.y'* holds. We only allow complex assignments if they always have a solution, i.e. there are always such values *x'* and *y'*. Both types of assignments are atomic actions. By definition,

$[\ \text{wlp.}(x := E).Q \equiv (x := E).Q\ ]$; and
$[\ \text{wlp.}(x, y := E, F).Q \equiv (x, y := E, F).Q\ ]$; and
$[\ \text{wlp.}(x : E.x).Q \equiv (\forall x' : E.x' : (x := x').Q)\ ]$.

In practice we rely on the following theorems:

$\{P\}\ x := E\ \{Q\} \equiv [\ P \Rightarrow (x := E).Q\ ]$; and
$\{P\}\ x : E.x\ \{Q\} \equiv [\ P \Rightarrow (\forall x' : E.x' : (x := x').Q)\ ]$.

1.2.3 The sequential composition

A composition like $S_0; S_1$ first executes statement S_0, and then statement S_1. By definition,

$$[\ wlp.(S_0; S1).R \equiv wlp.S_0.(wlp.S_1.R)\ ] .$$

In practice we rely on the following theorem:

$$\{P\}\ S_0; S_1\ \{R\} \Leftarrow \{P\}\ S_0\ \{Q\}\ \wedge\ \{Q\}\ S_1\ \{R\} \quad .$$

1.2.4 The alternative construct

For statements S_0 and S_1, and guards (i.e. boolean expressions) B_0 and B_1, we consider an alternative construct (often called a selection) to be denoted by IF, which is short for

if $B_0 \rightarrow S_0$ [] $B_1 \rightarrow S_1$ **fi** .

It contains the two guarded commands $B_0 \rightarrow S_0$ and $B_1 \rightarrow S_1$. An alternative construct like IF executes S_0 if B_0 evaluates to *true*, or S_1 if B_1 evaluates to *true*. The evaluation of a guard is an atomic action. By definition,

$$[\ wlp.IF.R \equiv (B_0 \Rightarrow wlp.S_0.R)\ \wedge\ (B_1 \Rightarrow wlp.S_1.R)\ ] \quad .$$

In practice we rely on the following theorem:

$$\{P\}\ IF\ \{R\} \equiv \{P \wedge B_0\}\ S_0\ \{R\}\ \wedge\ \{P \wedge B_1\}\ S_1\ \{R\} \quad .$$

1.2.5 The repetitive construct

For statements S_0 and S_1, and guards B_0 and B_1, we consider a repetitive construct (often called a repetition) to be denoted by DO, which is short for

do $B_0 \rightarrow S_0$ [] $B_1 \rightarrow S_1$ **od** .

A repetitive construct like DO is just a loop, which keeps executing S_0 or S_1 as long as the corresponding guard evaluates to *true*. When both guards are *false* the loop terminates. The evaluation of a guard is an atomic action.

In practice we rely on the Invariance Theorem:

$$\{P\}\ DO\ \{P \wedge \neg B_0 \wedge \neg B_1\} \Leftarrow \{P \wedge B_0\}\ S_0\ \{P\}\ \wedge\ \{P \wedge B_1\}\ S_1\ \{P\} \quad .$$

Sometimes we consider components that are intended to never terminate. Such non-terminating repetition is usually denoted as **[S]*, for *S* some statement.

1.3 Variables and grain-size

The state space of a multiprogram consists of the variables. We distinguish three kinds of variables:

- *local* (**loc** for short) variable of a component: a variable that may be changed and inspected by that component only;
- *private* (**priv** for short) variable of a component: a variable that may be changed by that component only and that may be inspected by all components;
- *shared* (**shar** for short) variable: a variable that may be changed and inspected by all components.

Sometimes we want to extend the state space locally. Therefore we can define local variables within inner blocks, like for fresh local variables *x* and *y*, and a statement *S*:

```
|[ var:  x, y
   S
]|
```

Related to variables, but not part of the state space, are the so-called constants. We distinguish two kinds of constants (**con** for short):

- *constant*: a "variable" that may not be changed and that may be inspected by all components;
- *constant* of a component: a "variable" that may not be changed and that may be inspected by only that component;

1.4 Grain-size

The actions we assume to be fine-grained enough in final solutions are the so-called one-point atomic actions. One-point atomic actions are guards or simple assignments that contain at most one reference to at most one private or shared variable. We assume that these one-point actions can be implemented atomically.

Apart from these traditional one-point atomic statements, we will also need one-point compare&swap statements. Therefore we now consider the compare&swap statement. Apart from the full version, we also consider a slightly lower-level version (which can easily be implemented using the full version).

For variables *x* and *r* and expressions *p* and *q* we define:

```
compare&swap(x, p, q, r)   ≡   < r := x
                                ; if x = p →  x := q
                                  [] x ≠ p →  skip
                                  fi >

compare&swap(x, p, q)      ≡   < if x = p →  x := q
                                  [] x ≠ p →  skip
                                  fi >
```

In a particular instantiation of the parameters, such statement is a one-point statement if all parameters together contain at most reference to at most one shared or private variable. We assume that the one-point compare&swap statement can be implemented atomically and we extend the one-point actions with the one-point compare&swap statement.

(One may now wonder whether the one-point versions are available as primitives on actual systems. For example, according to the specifications in [Int99], they can be implemented for 4-byte variables on the Intel486™ and for 8-byte variables on the Pentium® processor.)

1.5 Abbreviations

In our multiprograms we use the following abbreviations:

Abbreviation	Meaning
Con	Constraints
Inv	Invariants
Pre	Precondition
Prf	Proof obligations

1.6 Lemmata

The following lemmas are frequently used during program derivation:

- Orthogonality: An assertion that does not depend on a variable *x* is maintained by all assignments to *x*.

- Disjointness: Assertion *P* is maintained by *{Q} S* if $[\ P \wedge Q \ \Rightarrow\ \text{false}\ ]$.

- Private Variables: An assertion in a component that depends on private (or local) variables of that component only is maintained by all statements of other components.

- Widening: Assertion $x \leq y$ is maintained by "descents" of *x* and by "ascents" of *y*.

- Guard Strengthening Lemma: Program fragment $\{C \Rightarrow B\}$ **if** $B \rightarrow S$ **fi** may be replaced by $\{C \Rightarrow B\}$ **if** $C \rightarrow S$ **fi** without impairing the correctness of the annotation.

2. Private occurrences

Rule of Private Occurrences

> Occurrences of private variables of a component in expressions in elementary statements within that component can be considered as occurrences of local variables of that component.

End of Rule of Private Occurrences

Proof
Consider a private variable *x* of component C and an expression *E* in an elementary statement within that component containing variable *x*. We can consider variable *x* as a local variable in expression *E* if we can replace expression *E* by expression *E(x:=h)*, for *h* a fresh local variable of component C. Using the Rule of Leibniz, we therefore require precondition *h = x*.

We then consider that precondition *h = x* in component C. Global correctness is for free (Private Variables). Local correctness can be established by inserting one-point assignment *h := x*. Because variable *h* is a fresh variable, this assignment can not endanger other assertions.

Corollary
For private variable *x*, we consider assignment *x := x+1*. Note that variable *x* must be a private variable of the component in which this assignment occurs. This assignment can be implemented with one-point statements (Private Occurrences). For example, as described in the proof above, for fresh local variable *h* the following fragment does the job:

```
  h := x
; {h = x}
  x := h+1
```

3. An opportunity to eliminate global correctness proofs

This chapter describes a way to reduce the proof load for global correctness at the cost of a (potentially) stronger proof load for local correctness.

3.0 Specification

Consider the following part of a two-component multiprogram:

A: S_n *labelA*: ; S_{n+1}	B: T_n *labelB*: ; T_{n+1}

Computation proper

Component A contains at least the two consecutive statements S_n and S_{n+1} and component B contains at least the two consecutive statements T_n and T_{n+1}. We assume that statements S_n and T_n are atomic. Component A is at *labelA* when the execution of statement S_n has terminated, but the execution of S_{n+1} has not yet started; and similarly for component B at *labelB*.

We now consider the case that we have to ensure that some assertion X holds at the point that component A is at *labelA* and component B is at *labelB*:

A: S_n *labelA*: ; S_{n+1}	B: T_n *labelB*: ; T_{n+1}
Prf: **?** "component A at *labelA*" $\wedge$ "component B at *labelB*" $\Rightarrow$ X	

Specification

Such a proof obligation occurs in several different settings. For example, if statement S_{n+1} has precondition P, we could prove global correctness of assertion P under statement T_{n+1} by choosing $X \equiv wlp.T_{n+1}.P$. Or if statements S_{n+1} and T_{n+1} are guarded skips with guards C and D respectively, we could prove absence of total deadlock by choosing $X \equiv C \vee D$.

3.1 Solutions

This proof obligation can easily be fulfilled by requiring the validity of assertion X when component A is at *labelA* **or** when component B is at *labelB*. The consequence would be that we must prove global correctness of that assertion X. However, in general global correctness proofs can be huge or difficult. Therefore we will try a different approach.

Note that if component A is at *labelA* and component B is at *labelB*, one of those components was the last component that reached *labelA* or *labelB* respectively. Therefore we fulfil the proof obligation by requiring that both atomic statements S_n and T_n make X locally correct at *labelA* **and** at *labelB* respectively.

We propose to write an atomic statement and its postcondition for which we only have to prove local correctness within atomic brackets. Note that we may not use such postcondition as precondition of the next statement.

Thus we obtain:

A: $< S_n$ {**?** X} $>$; S_{n+1}	B: $< T_n$ {**?** X} $>$; T_{n+1}

Solution

An afterthought

Our specification was rather restrictive in the sense that we considered only two components and the two statements we considered in those two components were consecutive. This allowed us to focus on the essence of the problem. We can eliminate these restrictions at the cost of proving maintenance of X under the statements in the extra components and the extra statements between statements S_n and S_{n+1} and between statements T_n and T_{n+1}.

4. Definitions of non-blocking multiprograms

This chapter investigates some existing definitions of non-blocking multiprograms and proposes new definitions.

4.0 Some existing definitions

Before we mention a couple of definitions of non-blocking multiprograms as common in the literature, it is important to realise that they also consider a computational model in which halting failures may occur (see section 1.0).

Unfortunately, some of the definitions in the literature conflict with each other. Therefore we consider a consistent subset of these definitions. Most definitions of *wait-free* multiprograms are based on the following definition:

> "A *wait-free* implementation (…) is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes." ([Her91])

Sometimes the wider class of *lock-free* multiprograms is also defined:

> "Two basic *non-blocking* methods have been proposed in the literature, *lock-free* and *wait-free*. *Lock-free* implementations (...) guarantee that at any point in time in any possible execution some operation will complete in a finite number of steps. (...) In *wait-free* implementations each task is guaranteed to correctly complete any operation in a bounded number of its own steps, regardless of overlaps and the execution speed of other processes (…) ." ([ST00])

Unfortunately, these definitions are rather operational. A more syntactically oriented definition is:

> "In a *wait-free* object implementation, operations must be implemented using bounded, sequential code fragments, with no blocking synchronisation constructs." ([AJR97])

4.1 Alternative definitions

We do not want to reason explicitly about halting components and possible executions, so we have to find new definitions that can be used conveniently like the syntactic definition of [AJR97].

First observe that there are essentially two constructs that can hinder progress of a component:

- blocking statements (for example the guarded skip and the P-operation);
- repetitions (for example recursion and busy or dynamic waiting).

We then define two disjoint classes of multiprograms:

- A component is a *lock-based* component if and only if that component contains at least one blocking statement.

- A component is a *non-blocking* component if and only if that component contains no blocking statements.

- A multiprogram is a *lock-based* multiprogram if and only if at least one of its components is a *lock-based* component.

- A multiprogram is a *non-blocking* multiprogram if and only if all of its components are *non-blocking* components.

Because critical sections, which are common in *lock-based* multiprograms, are undesired in a computational model in which halting failures may occur, we want to consider *non-blocking* multiprograms. Although *lock-based* multiprograms can be transformed into *non-blocking* multiprograms using busy waiting, that transformation does not eliminate the critical sections. We want to define some more useful classes of *non-blocking* multiprograms by restricting the use of repetitions. We define two classes of *non-blocking* multiprograms:

- A set of *non-blocking* components is a *lock-free* set of components if and only if for each non-empty set of repetitions from all components, termination of at least one repetition is guaranteed in that set of components.

- A *non-blocking* component is a *wait-free* component in a multiprogram if and only if for each repetition from that component, termination is guaranteed in that multiprogram.

- A *non-blocking* multiprogram is a *lock-free* multiprogram if and only if each subset of its components is a *lock-free* set of components.

- A *non-blocking* multiprogram is a *wait-free* multiprogram if and only if all of its components are *wait-free* components in that multiprogram.

4.2 Some Properties

Using these definitions we have for example the following properties:

- Each set of *wait-free* components in a multiprogram is a *lock-free* set of components.
- Each *wait-free* multiprogram is a *lock-free* multiprogram.

- In each *non-blocking* multiprogram there is no danger of (individual) deadlock.
- Each *wait-free* multiprogram does not suffer from individual starvation.

5. Strategies to derive non-blocking multiprograms

This chapter describes our strategy to derive non-blocking multiprograms.

5.0 Non-blockingness

A non-blocking multiprogram may not contain blocking statements. Note that the Guarded Command Language contains only one potentially blocking statement, namely the selection. Thus the easiest way to obtain a non-blocking multiprogram, is to avoid using selections (see e.g. the “Non-blocking write protocol” (chapter 6)).

If we want to, or have to, use a selection, we must ensure that we use a non-blocking instance. A selection like **if** $B_0 \rightarrow S_0$ [] $B_1 \rightarrow S_1$ **fi** is non-blocking if its pre-assertion implies that at least one of the guards is *true*, i.e. $B_0 \vee B_1$.

If we encounter, during a derivation, a potentially blocking statement for which we cannot prove the required precondition, we have to adapt that statement. For a selection this means that we must weaken a guard or that we must add extra guarded commands. Because weakening a guard is in general not allowed, we may only add extra guarded commands.

For the main application of this technique, see section 5.2.

5.1 Progress

The progress properties of a non-blocking multiprogram are specified by the subclasses of non-blockingness: lock-freeness and wait-freeness. Note that the Guarded Command Language contains only two constructions that can endanger progress: repetition and recursion. Thus the easiest way to obtain a wait-free (and hence lock-free) solution, is to avoid using these constructions (see e.g. the “Coarse-grained wait-free handshake register” (chapter 9) and the “Fine-grained wait-free handshake register” (chapter 10)).

If we want to, or have to, use such constructions, we must ensure that progress according to lock-freeness or wait-freeness is guaranteed (for their definitions see chapter 4). For examples of these techniques, see the derivations of the “Lock-free small data structures” (see chapter 8) and the “Wait-free write-all problem” (see chapter 11).

5.2 Universal constructions

As mentioned in [Her91] and [Her93], many synchronisation problems have neither a wait-free nor a lock-free solution using only traditional one-point statements while they have a wait-free and a lock-free solution using a so-called universal construction. Two examples of universal constructions are the compare&swap statement and the *load_linked - store_conditional* statements. Whereas [Her93] has found the *load_linked - store_conditional* statements easy to use, we have easily derived solutions that use the compare&swap statement because the compare&swap statement can easily be defined and manipulated in our method of multiprogramming.

The following situation usually gives rise to the use of compare&swap statements. Suppose we are dealing with an assertion *Q* for which we can establish local correctness by a statement *S* if we require statement *S* to have precondition *P*:

{**?** P}
S
{**?** Q}

Suppose it is difficult to get assertion *P* globally correct. A simple solution to get assertion *P* globally (and locally) correct is to embed statement *{P} S* in an atomic guarded statement with guard *P*:

if < P → {P} S > **fi**
{**?** Q}

However, in general this is a blocking statement, while we want to obtain a non-blocking solution. As described in section 5.0, we can solve it by adding *P* as pre-assertion of this selection, but that would return us to our original problem of getting assertion *P* globally correct. So we add an extra guarded statement *P' → S'* to the selection, for which we assume *{P'} S' {Q}*, and we require the selection to have pre-assertion *P ∨ P'*:

{**?** P ∨ P'}
if < P → {P} S >
[] P' → {**?** P'} S'
fi
{**?** Q}

However, in general this selection is considered to be too coarse-grained. Note that it resembles a compare&swap statement. Fortunately, in all cases we encountered such construction, we could implement it using a compare&swap statement.

For a nice example of this technique, see the “Wait-free consensus protocol” (see chapter 7). For more examples, see also the “Lock-free small data structures” (see chapter 8) and the “Wait-free write-all problem” (see chapter 11).

Derivations

6. Non-blocking write protocol

This chapter describes a fine-grained implementation of a non-blocking write protocol using boolean variables only.

6.0 Specification

Consider the following two-component multiprogram from chapter 22 (The Non-Blocking Write Protocol) of [FvG99]:

Pre: f = 0 ∧ g = 0 ∧ y < x	
W: **priv**: f, g * [f := f+1 ; WRITE ; g := g+1]	R: **loc**: x, y **do** ¬(x ≤ y) → y := g ; READ ; x := f **od**

Computation Proper

Wait-free component W repeatedly writes information into an anonymous data structure by the non-atomic WRITE statement. In between two successive WRITEs the data structure is assumed to be consistent. Non-blocking component R repeatedly reads information from the data structure by the non-atomic READ statement.

The protocol guarantees that component R terminates if and only if component R is not halted and the last execution of the body of component R did not interfere with executions of the body of component W. Thus upon termination of component R, during the last READ statement a consistent data structure has been read.

Observe that in this algorithm variables *f* and *g* are unbounded integer variables and so are variables *x* and *y*. We want to obtain an implementation in terms of boolean variables (and one-point atomic statements) only.

6.1 Towards boolean variables

As a starting point we use guard $\neg(x \leq y)$ of the repetition in component R. If we want to eliminate variables *x* and *y*, this guard must be changed. Because the guard of the repetition is not used as precondition of the body of the repetition, we may, analogous to the Guard Strengthening Lemma, weaken that guard without impairing partial correctness. Note that we then have to take care of termination of component R. A solution (using zero boolean variables) with guard *true* is definitely not appropriate.

We introduce a fresh shared variable *c* with invariant $c \Rightarrow x \leq y$ of the repetition in component R. Then we weaken the guard into $\neg c$. It turns out that if we do not weaken the guard at this point, we will have to do it later on, which involves much extra work.

We now deal with that repetition invariant $c \Rightarrow x \leq y$ in component R. Global correctness will be guaranteed in what follows. For initial correctness we must require precondition $\neg c$ due to precondition $y < x$. For invariance we require precondition $c \Rightarrow f \leq y$ of assignment $x := f$ in component R.

<table>
<tr><td colspan="2">Pre: f = 0 ∧ g = 0 ∧ y < x ∧ ¬c
shar: c</td></tr>
<tr><td>W: priv: f, g
* [f := f+1
; WRITE
; g := g+1
]</td><td>R: loc: x, y
{Inv.: c ⇒ x ≤ y}
do ¬c →
y := g
; READ
; {? c ⇒ f ≤ y}
x := f
od</td></tr>
</table>

We now deal with assertion $c \Rightarrow f \leq y$ in component R. For global correctness we calculate:

(f := f+1).(c ⇒ f ≤ y)
≡ { substitution }
c ⇒ f < y
⇐ { calculus, invariant $y \leq g$ (Widening) and invariant $g \leq f$ (Topology) are system invariants (if they hold initially) and therefore $y \leq f$ }
¬c

We extend the assignment $f := f+1$ in component W with an assignment $c := false$. Note that this assignment does not endanger other assertions (Widening).

For local correctness of assertion $c \Rightarrow f \leq y$ in component R, note that the READ statement does not affect that assertion. Then we calculate:

(y := g).(c ⇒ f ≤ y)
≡ { substitution }
c ⇒ f ≤ g

We could require this condition as precondition of assignment $y := g$ in component R. Local correctness would follow from the guard of the repetition and global correctness would be guaranteed. However, we would then obtain a solution (with <u>one</u> boolean variable) in which variable c is stably *false* and consequently component R can not terminate because its guard is stably *true*. Therefore we will strengthen that precondition:

c ⇒ f ≤ g
⇐ { introduce fresh private variable b in component W, variables f and g of component W must also be eliminated }
(c ⇒ b) ∧ (b ⇒ f ≤ g)

The first conjunct is catered for by extending assignment $y := g$ with an assignment $c := b$. We adopt the second conjunct as invariant P0, because it contains only variables of component W.

Pre: f = 0 ∧ g = 0 ∧ y < x ∧ ¬c **shar**: c	
W: **priv**: b, f, g * [f, c := f+1, false ; WRITE ; g := g+1]	R: **loc**: x, y {Inv.: c ⇒ x ≤ y} **do** ¬c → y, c := g, b ; {c ⇒ f ≤ y} READ ; {c ⇒ f ≤ y} x := f **od**
Inv.: P0: **?** b ⇒ f ≤ g	

We are left with invariant P0: $b \Rightarrow f \leq g$. Although initial correctness is guaranteed by precondition $f = 0 \wedge g = 0$, we require b for progress reasons. For maintenance we will use the topology of component W. We establish precondition $\neg b$ of assignment $f, c := f+1, false$ in component W by inserting an assignment $b := false$. For progress reasons, we extend assignment $g := g+1$ in component W with an assignment $b := true$.

Thus we obtain:

Pre: f = 0 ∧ g = 0 ∧ y < x ∧ ¬c ∧ b **shar**: c	
W: **priv**: b, f, g * [b := false ; {¬b} f, c := f+1, false ; WRITE ; g, b := g+1, true]	R: **loc**: x, y {Inv.: c ⇒ x ≤ y} **do** ¬c → y, c := g, b ; {c ⇒ f ≤ y} READ ; {c ⇒ f ≤ y} x := f **od**
Inv.: P0: b ⇒ f ≤ g	

If we would now eliminate variables f, g, x and y, we would obtain an implementation using the two boolean variables b and c. Unfortunately, that implementation would contain non-one-point assignment $c := b$.

6.2 Grain-size

We now deal with non-one-point assignment $c := b$ as part of assignment $y, c := g, b$ in component R. Observe that variable c is only inspected in component R, which is the same component as in which that assignment occurs. Unfortunately, we have also assignment $c := false$ in component W.

We introduce fresh shared variable *d* and fresh local variable *e* in component R to take over the rôle of shared variable *c*. Then we make variable *c* superfluous by requiring invariant P1: $d \wedge e \Rightarrow c$ and weakening the guard into $\neg(d \wedge e)$. It turns out again that if we do not weaken the guard at this point, we will have to do it later on, which involves much extra work.

We now deal with that invariant P1: $d \wedge e \Rightarrow c$. For initial correctness we have to require precondition $\neg(d \wedge e)$ due to precondition $\neg c$. To guarantee maintenance under assignment *f, c := f+1, false* in component W and assignment *y, c := g, b* in component R, we want to extend them with a one-point assignment to variable *d* or *e*. Because variable *e* is a local variable of component R, we have to extend assignment *f, c := f+1, false* in component W with assignment *d := false*. Because variable *d* is a shared variable and variable *b* is a private variable of component W, we have to extend assignment *y, c := g, b* in component R with one-point assignment *e := b*.

Thus we obtain:

<table>
<tr><td colspan="2">Pre: f = 0 ∧ g = 0 ∧ y < x ∧ ¬c ∧ b ∧ ¬(d ∧ e)
  shar: c, d</td></tr>
<tr><td>W: priv: b, f, g
 * [b := false
  ; {¬b}
   f, c, d := f+1, false, false
  ; WRITE
  ; g, b := g+1, true
  ]</td><td>R: loc: e, x, y
 {Inv.: c ⇒ x ≤ y}
 do ¬(d ∧ e) →
   y, c, e := g, b, b
  ; {c ⇒ f ≤ y}
   READ
  ; {c ⇒ f ≤ y}
   x := f
 od</td></tr>
<tr><td colspan="2">Inv.: P0: b ⇒ f ≤ g
  P1: d ∧ e ⇒ c</td></tr>
</table>

If we would now eliminate variables *c, f, g, x* and *y*, we would obtain a fine-grained implementation using the three boolean variables *b, d* and *e*. So we obtained a fine-grained implementation at the cost of one extra boolean variable.

6.3 Progress

In this section we consider the case that component R is not halted and not yet terminated, and component W halts outside the body of its repetition. Note that then variable *b* is stably *true* (Private Variables). In that case we want to enable component R to terminate, so we must ensure that if component R is not yet terminated, guard $\neg(d \wedge e)$ of the repetition in component R eventually becomes stably *false*. Therefore we have to ensure that both variables *d* and *e* eventually become stably *true*.

Assignment *e := b* in the body of the repetition of component R, which reduces to assignment *e := true* because variable *b* is stably *true*, ensures that eventually variable *e* becomes stably *true*.

Unfortunately, component R contains no assignments to variable *d*, so if ¬*d* holds, component R will not terminate. Therefore we have to insert an assignment *d := true* in the body of the repetition in component R. For maintenance of invariant P1, such assignment must guarantee e ⇒ c. We could do so by extending that assignment with an assignment *c := true*. Because assignment *y, c, e := g, b, b* is the only statement in component R whose precondition consists only of the system invariants, we insert assignment *c, d := true, true* before assignment *y, c, e := g, b, b*.

Thus we obtain:

Pre: f = 0 ∧ g = 0 ∧ y < x ∧ ¬c ∧ b ∧ ¬(d ∧ e) **shar**: c, d	
W: **priv**: b, f, g * [b := false ; {¬b} f, c, d := f+1, false, false ; WRITE ; g, b := g+1, true]	R: **loc**: e, x, y {Inv.: c ⇒ x ≤ y} **do** ¬(d ∧ e) → c, d := true, true ; y, c, e := g, b, b ; {c ⇒ f ≤ y} READ ; {c ⇒ f ≤ y} x := f **od**
Inv.: P0: b ⇒ f ≤ g P1: d ∧ e ⇒ c	

We now eliminate variables *c*, *f*, *g*, *x* and *y* and remove all annotation. Thus we obtain the following implementation:

Pre: b ∧ ¬(d ∧ e) **shar**: d	
W: **priv**: b * [b := false ; d := false ; WRITE ; b := true]	R: **loc**: e **do** ¬(d ∧ e) → d := true ; e := b ; READ **od**

We obtained a fine-grained implementation of “The Non-blocking Write Protocol” using only three boolean variables instead of the original four unbounded integer variables. Note that although variable *d* is a shared variable, all communication is in fact unidirectional from component W to component R, because component W does not inspect any variable.

7. Wait-free consensus protocol

This chapter describes derivations of implementations of a wait-free consensus protocol that use the high-level primitive compare&swap.

7.0 Specification

A consensus protocol is a protocol that guarantees that a set of components, each with one input value, eventually agree on one of their input values.

To develop a formal specification, we consider a set of components *C*. As a convention, for dummy variable *c* we always require $c \in C$. Each component *c* has an input value *x.c* and a variable *y.c* to return the value the components agreed on. We consider two (functional) requirements and three (non-functional) constraints.

The first requirement we consider is that the protocol must be consistent: all components compute the same value. Therefore we introduce shared variable *v* to indicate that common value. We require that finally output variable *y.c* be equal to that variable *v*.

	shar: v
Comp.c:	**con**: x.c **loc**: y.c … {**?** y.c = v}

The second requirement we consider is that the protocol must be valid: the common value must be the input value of one of the components. Therefore we introduce shared specification variable *X* to indicate a set of input values. Initially set *X* is empty and each component adds its input value *x.c* to set *X*. We require that finally variable *v* is an element of that set *X*.

Pre:	X = ∅ **shar**: v, X
Comp.c:	**con**: x.c **loc**: y.c X := X ∪ {x.c} … {**?** y.c = v}{**?** v ∈ X}

Specification

The problem thus specified has to be solved under three additional constraints, to wit

- that the final algorithm be symmetric in the components (for fairness reasons);
- that the final algorithm be entirely expressed in one-point atomic statements;
- that the final algorithm be wait-free (see chapter 4).

7.1 Skeleton Version

We first deal with assertion $y.c = v$ in component c. To establish local correctness without endangering assertion $y.d = v$ in components $d: d \neq c$, we insert an assignment $y.c := v$. Global correctness of assertion $y.c = v$ will be guaranteed in what follows.

Global correctness of assertion $v \in X$ in component c is for free (Widening). For local correctness of that assertion we require that assertion as precondition of assignment $y.c := v$.

Pre:	X = ∅ **shar**: v, X
Comp.c:	**con**: x.c **loc**: y.c X := X ∪ {x.c} … {**?** v ∈ X} y.c := v {y.c = v}{v ∈ X}

We are left with queried assertion $v \in X$ in component c. For establishing local correctness without breaking symmetry, we have, apart from guarded command $v \in X \rightarrow$ ***skip*** (to be addressed later), hardly any choice but inserting an assignment $v := x.c$ for which we require precondition $x.c \in X$. Global correctness of assertion $v \in X$ is for free (Widening). Because assignment $v := x.c$ can endanger assertion $y.d = v$ (with co-assertion $v \in X$) in components $d: d \neq c$, we calculate:

|[y.d = v ∧ v ∈ X
▷ (v := x.c).(y.d = v)
≡ { substitution }
y.d = x.c
≡ { target assertion $y.d = v$, eliminate variables of component d }
v = x.c
⇐ { calculus, condition $v = x.c$ makes no sense as co-assertion in component d (symmetry constraint) or as precondition of assignment $v := x.c$ }
false
≡ { disjointness, use co-assertion $v \in X$ }
v ∉ X
]|

We adopt this condition as precondition of assignment $v := x.c$.

Pre:	X = ∅ **shar**: v, X
Comp.c:	**con**: x.c **loc**: y.c X := X ∪ {x.c} … {**?** x.c ∈ X}{**?** v ∉ X} v := x.c {v ∈ X} y.c := v {y.c = v}{v ∈ X}

We first deal with pre-assertion $v \notin X$ of assignment $v := x.c$ in component c. For establishing local correctness without endangering assertions $v \in X$ in components $d: d \neq c$, we have hardly any choice but embedding statement {**?** x.c ∈ X}{**?** v ∉ X} $v := x.c$ in a selection statement with guard $v \notin X$:

if v ∉ X → {**?** x.c ∈ X}{**?** v ∉ X} v := x.c **fi**

For global correctness of that assertion $v \notin X$ under assignment $v := x.d$ in components $d: d \neq c$, we place that guarded command within atomic brackets and require assertion $x.c \in X$ as precondition of the selection statement.

{**?** x.c ∈ X}
if < v ∉ X → {x.c ∈ X}{v ∉ X} v := x.c > **fi**

Because the components must be wait-free, we must ensure that this selection statement is not a blocking statement. Recall that we when we introduced assignment $v := x.c$ to establish local correctness of assertion $v \in X$, we mentioned as alternative guarded command $v \in X \rightarrow$ {v ∈ X} ***skip***. We extend the selection statement with this guarded command.

{**?** x.c ∈ X}
if < v ∉ X → {x.c ∈ X}{v ∉ X} v := x.c >
[] v ∈ X → {v ∈ X} **skip**
fi

We are left with assertion $x.c \in X$. Global correctness is for free (Widening). Local correctness follows from preceding assignment $X := X \cup \{x.c\}$.

Thus we obtain:

Pre:	X = ∅ **shar**: v, X
Comp.c:	**con**: x.c **loc**: y.c X := X ∪ {x.c} ; {x.c ∈ X} **if** < v ∉ X → {x.c ∈ X}{v ∉ X} v := x.c > [] v ∈ X → {v ∈ X} **skip** **fi** ; {v ∈ X} y.c := v {y.c = v}{v ∈ X}

Skeleton Version

According to Theorem 2 of [Her91], there exists no wait-free implementation for more than one component using “read / write registers” (in our terminology: one-point assignments) only. Therefore we will investigate how to implement this algorithm using higher-level primitives.

7.2 Compare&Swap Implementation

According to Theorem 5 of [Her91], there exists a consensus protocol using "compare&swap registers" for any number of components. The atomic selection statement in our algorithm resembles a compare&swap statement (in particular the second version presented in section 1.4). Before we can implement our algorithm using that primitive, we must match the guards of our selection statement with the guards of that primitive. Therefore, we introduce fresh constant $\perp$ with invariant $v \in X \equiv v \neq \perp$.

We now deal with invariant $v \in X \equiv v \neq \perp$. For initial correctness we must require precondition $v = \perp$ due to precondition $X = \emptyset$. For maintenance we calculate:

$(X := X \cup \{x.c\}).(v \in X \equiv v \neq \perp)$
$\equiv$ { substitution }
$v \in X \cup \{x.c\} \equiv v \neq \perp$
$\equiv$ { set calculus }
$v \in X \vee v = x.c \equiv v \neq \perp$
$\equiv$ { invariant $v \in X \equiv v \neq \perp$ }
$v \neq \perp \vee v = x.c \equiv v \neq \perp$
$\equiv$ { calculus }
$v = x.c \Rightarrow v \neq \perp$
$\Leftarrow$ { calculus, eliminate variable v }
$x.c \neq \perp$

We require this condition as requirement on the value of constant $x.c$.

$|[\; x.c \in X \wedge x.c \neq \perp$
$\triangleright$ $(v := x.c).(v \in X \equiv v \neq \perp)$
$\equiv$ { substitution }
$x.c \in X \equiv x.c \neq \perp$
$\equiv$ { precondition $x.c \in X$ }{ use requirement $x.c \neq \perp$ of constant $x.c$ }
true
$]|$

Using constant $\perp$ we do not need specification variable X anymore, so we eliminate variable X. Thus we obtain:

Pre:	$v = \perp$ **shar**: v
Comp.c:	**con**: x.c: $x.c \neq \perp$ **loc**: y.c **if** < $v = \perp \rightarrow$ {$v = \perp$} v := x.c > [] $v \neq \perp \rightarrow$ {$v \neq \perp$} **skip** **fi** ; {$v \neq \perp$} y.c := v {y.c = v}{$v \neq \perp$}

If we would now implement the atomic selection statement using the compare&swap primitive and remove all annotation, we would obtain:

Pre:	v = ⊥ **shar**: v
Comp.c:	**con**: x.c: x.c ≠ ⊥ **loc**: y.c compare&swap(v, ⊥, x.c) ; y.c := v

When we now compare this result with that Theorem 5 of [Her91], one may wonder whether "compare&swap register" *v* may be inspected as in assignment *y.c := v*. Because we can implement assignment *y.c := v* by statement *compare&swap(v, z, z, y.c)* for any value *z*, we do not have to worry about this. Thus in this section we obtained a proof of that theorem.

7.3 Another Compare&Swap Implementation

In [Her91] another implementation is used to prove his Theorem 5. In that implementation variable *v* occurs in one compare&swap statement only. Therefore we now try to implement assignment *y.c := v* such that it does not depend on variable *v*.

Observe that the value of *v* determines whether the atomic selection statement changes the value of *v*. So only the first component that executes the atomic selection statement finds initial value ⊥ and changes the value of *v*. Therefore we will try to eliminate the occurrence of variable *v* in assignment *y.c := v* by using the value of *v* at the beginning of the atomic selection statement. Then we will need the full version of the compare&swap primitive.

We introduce fresh local variable *r.c* in component *c* and replace assignment {v ≠ ⊥} *y.c := v* as follows (Leibniz):

Pre:	v = ⊥ **shar**: v
Comp.c:	**con**: x.c: x.c ≠ ⊥ **loc**: r.c, y.c **if** < v = ⊥ → {v = ⊥} v := x.c > [] v ≠ ⊥ → {v ≠ ⊥} **skip** **fi** ; {v ≠ ⊥} **if** r.c = ⊥ → {v ≠ ⊥}{**?** v = x.c} y.c := x.c [] r.c ≠ ⊥ → {v ≠ ⊥}{**?** v = r.c} y.c := r.c **fi** {y.c = v}{v ≠ ⊥}

We now deal with both assertions *v = x.c* and *v = r.c* in component *c*. Global correctness is for free (Disjointness). For local correctness we require the second selection statement to have preconditions *r.c ≠ ⊥ ∨ v = x.c* and *r.c = ⊥ ∨ v = r.c*.

We then deal with both assertions $r.c \neq \bot \ \vee \ v = x.c$ and $r.c = \bot \ \vee \ v = r.c$ in component c. Thanks to co-assertion $v \neq \bot$, global correctness is for free (Disjointness). Local correctness is guaranteed if we require the first selection statement to have precondition $v = r.c$.

We now deal with precondition $v = r.c$ of the selection statement. For establishing local correctness without endangering other assertions, we insert an assignment $r.c := v$; for global correctness of assertion $v = r.c$ we insert this assignment within the atomic brackets of the first selection statement.

Thus we obtain:

Pre:	v = ⊥ **shar**: v
Comp.c:	**con**: x.c: x.c ≠ ⊥ **loc**: r.c, y.c < r.c := v ; {v = r.c} **if** v = ⊥ → {v = r.c}{v = ⊥} v := x.c [] v ≠ ⊥ → {v = r.c}{v ≠ ⊥} **skip** **fi** > ; {v ≠ ⊥}{r.c ≠ ⊥ ∨ v = x.c}{r.c = ⊥ ∨ v = r.c} **if** r.c = ⊥ → {v ≠ ⊥}{v = x.c} y.c := x.c [] r.c ≠ ⊥ → {v ≠ ⊥}{v = r.c} y.c := r.c **fi** {y.c = v}{v ≠ ⊥}

We now implement the atomic selection statement using the full version of the compare&swap primitive and remove all annotation:

Pre:	v = ⊥ **shar**: v
Comp.c:	**con**: x.c: x.c ≠ ⊥ **loc**: r.c, y.c compare&swap(v, ⊥, x.c, r.c) ; **if** r.c = ⊥ → y.c := x.c [] r.c ≠ ⊥ → y.c := r.c **fi**

This implementation is used in [Her91] to prove his Theorem 5.

8. Lock-free small data structures

This chapter describes a lock-free implementation of small shared data structures based on the "Single Word Protocol" of [Her90]. This protocol lies at the heart of the other lock-free protocols of [Her90] and in fact also of the lock-free protocols of [Her93].

8.0 Specification

A protocol for a shared data structure guarantees that a set of primitive operations on that data structure maintain consistency of that data structure when these operations are executed concurrently. We consider small data structures that can be appropriately modelled as single variables.

To develop a formal specification, we represent the shared data structure as a fresh shared variable *X* of a data type, and the primitive operations as a set *F* of functions from and to that data type. Then we consider a set of components *C*. As a convention, for dummy variable *c* we always require $c \in C$. Each component repeatedly performs a primitive operation on that shared data structure.

```
                con:   F
                shar:  X
Comp.c:     * [ var:   f
                f: f ∈ F
              ; X := f.X
              ]
```

Computation Proper

The problem thus specified has to be solved under two additional constraints, to wit

- that the final algorithm be entirely expressed in one-point atomic statements;
- that the final algorithm be lock-free (see chapter 4).

8.1 Skeleton Version

Observe that the multiprogram contains no queried items and it is lock-free (actually it is even wait-free). It only contains non-one-point assignment *X := f.X*. We have hardly any choice but splitting this assignment into two one-point assignments.

```
                con:   F
                shar:  X
Comp.c:     * [ var:   f, x
                f: f ∈ F
              ; x := X
              ; {? x = X}
                X := f.x
              ]
```

Then we are left with assertion $x = X$, for which local correctness is guaranteed by construction. Unfortunately, global correctness can be endangered by an assignment to X in another component. Therefore we decide to place assertion $x = X$ and assignment $X := f.x$ within atomic brackets. Then we must also establish local correctness of that assertion within those atomic brackets. Because using preceding assignment $x := X$ to establish local correctness would return us to the original problem, we introduce a selection with guard $x = X$.

```
                con:   F
                shar:  X
Comp.c:     * [ var:   f, x
                f: f ∈ F
              ; x := X
              ; if < x = X →   {x = X} X := f.x > fi
              ]
```

Because the multiprogram must be lock-free and hence non-blocking, we must also add a guarded statement with a guard weaker than $x \neq X$. In case $x \neq X$ we have hardly any choice but recursively re-establishing $x = X$. By introducing recursion, lock-freeness may again be endangered. Observe that if $x \neq X$ holds in component c, some component must have executed an assignment to X after the last assignment $x := X$ in component c and hence must have terminated. So lock-freeness is guaranteed.

```
                con:   F
                shar:  X
Comp.c:     * [ var:   f
                f: f ∈ F
              ; apply.f
              ]
apply( f ) =  |[ var:   x
                x := X
              ; if < x = X →   {x = X} X := f.x >
                []   x ≠ X →   apply( f )
                fi
              ]|
```

Skeleton Version

8.2 Implementation

We are left with the implementation of the guarded command within atomic brackets. Note that the selection looks like the compare&swap primitive (see [AM5]) with as main difference that the second guarded command should only contain a **skip**. To enhance this similarity we rewrite our program as follows:

```
                con:   F
                shar:  X
Comp.c:     * [ var:   f
                f: f ∈ F
              ; apply.f
              ]
apply( f ) = |[ var:   x
                x := X
              ; if < x = X →   {x = X} X := f.x > ; skip
                []   x ≠ X →             skip       ; apply( f )
                fi
              ]|
```

To obtain a real compare&swap, we will try to “distribute the selection over both semi-colons in the selection”. But before we can do so, we introduce fresh local variable *r* as follows:

```
                con:   F
                shar:  X
Comp.c:     * [ var:   f
                f: f ∈ F
              ; apply.f
              ]
apply( f ) = |[ var:   r, x
                x := X
              ; < r := X
                ; {r = X}
                  if x = X →   {x = r}{x = X} X := f.x > ; {x = r} skip
                  [] x ≠ X →   {x ≠ r}        skip     > ; {x ≠ r} apply( f )
                  fi
              ]|
```

Because assertions $x = r$ and $x \neq r$ are mutually disjunct and globally correct (Private Variables), we can safely perform the distribution. Note that we need local variable *r*, because although assertions $x = X$ and $x \neq X$ are also mutually disjunct, they are not globally correct outside the atomic brackets.

```
                con:   F
                shar:  X
Comp.c:     * [ var:   f
                f: f ∈ F
              ; apply.f
              ]
apply( f ) = |[ var:   r, x
                x := X
              ; < r := X
                ; {r = X}
                  if x = X →   {x = r}{x = X} X := f.x
                  [] x ≠ X →   {x ≠ r}        skip
                  fi
                >
              ; if x = r →     {x = r} skip
                [] x ≠ r →     {x ≠ r} apply( f )
                fi
              ]|
```

The statements within atomic brackets can now be implemented by a one-point compare&swap statement. To obtain a cleaner program, we transform the recursion into a repetition.

Thus we obtain:

```
                con:   F
                shar:  X
Comp.c:     * [ var:   f, r, x
                f: f ∈ F
              ; r, x: x ≠ r
              ; do x ≠ r →     x := X
                               ; compare&swap(X, x, f.x, r)
                od
              ]
```

Lock-free small data structure

9. Coarse-grained wait-free handshake register

This chapter describes a derivation of a coarse-grained algorithm for a wait-free handshake register based on the fine-grained algorithm described in [Hes98].

9.0 Specification

A handshake register controls access to a data structure shared by a writer and a reader. The writer repeatedly writes information into the data structure. The reader repeatedly reads information from the data structure. Writing into the data structure should not overlap with some other writing into or reading from the data structure.

One may wonder whether non-overlap and wait-freeness are conflicting requirements. If we are prepared to introduce multiple copies of the shared data structure, they do not have to conflict. For instance if we ensure that writing and reading occurs in different copies.

To develop a formal specification, we consider the following two-component multiprogram:

Pre: i = 0	
W: **priv**: i, x * [WRITE(x.i) ; i := i+1]	R: **loc**: s * [READ(x.s)]

Computation Proper

Component W writes successive versions of the shared data structure by filling private array *x*. Private variable *i* indicates the number of written elements so that *x[0..i)* is written. Component R reads versions of the shared data structure by reading element *s* from array *x*, with *s* a local variable.

Now we can specify how these components have to be synchronised. We consider three (functional) requirements and one (non-functional) constraint.

The first requirement we consider is that the values read by component R are written values from array *x*. To specify this, we require that *s* is less than *i* as precondition of the READ statement. Note that this precondition implies that for each element of *x* reading and writing do not overlap.

Pre: i = 0	
W: **priv**: i, x * [WRITE(x.i) ; i := i+1]	R: **loc**: s * [… {**?** s < i} READ(x.s)]

The next requirement we consider is recentness. Recentness means that the value read by component R was at some time during that read phase the most recently written value. To specify this, we introduce fresh local variable *m* in component R to store, at the beginning of the read phase, the index of the element of *x* that was most recently written in the shared data structure as indicated by variable *i*. We require that *s* is at least *m* as precondition of the READ statement.

Pre: i = 0	
W: **priv**: i, x * [WRITE(x.i) ; i := i+1]	R: **loc**: m, s * [m := i-1 … ; {**?** s < i}{**?** m ≤ s} READ(x.s)]

The last requirement we consider is sequentiality. Sequentiality means that for two read phases, the value read during the later phase is not older than the value read during the previous phase. Therefore we require as constraint C0 that the value of s is ascending in time.

Pre: i = 0	
W: **priv**: i, x * [WRITE(x.i) ; i := i+1]	R: **loc**: m, s * [m := i-1 … ; {**?** s < i}{**?** m ≤ s} READ(x.s)]
Con: C0: **?** *s* is ascending in time	

Specification

The problem thus specified has to be solved under one additional constraint, to wit that the final algorithm be wait-free.

9.1 Skeleton solution

We first deal with both queried assertions in component R at once: $m \leq s < i$. Global correctness is for free (Widening). For local correctness, we try to add an assignment $s := E$ in component R with E some expression (to be determined). We then have to prove precondition $m \leq E < i$ of that assignment.

We will now investigate the consequences of the two simple choices for expression E, that are $E = m$ and $E = i-1$. Note that we can not guarantee that one of these choices will lead us to the algorithm described in [Hes98].

These two expressions are equivalent in the sense that in both cases the weakest (liberal) precondition of assertion $m \leq s < i$ under assignment $s := E$ is $m < i$. For constraint C0 we also have to require precondition $s \leq E$ of assignment $s := E$. For the two choices for E this means $s \leq m$ or $s < i$ respectively. For proving local correctness of precondition $s \leq m$ of assignment $s := m$ we would have to require $s < i$ as precondition of assignment $m := i-1$, so we can better directly chose $E = i-1$.

Thus to establish local correctness of assertions $m \leq s$ and $s < i$, we insert an assignment *s := i-1* for which we require precondition $m < i$. For maintenance of constraint C0 we require invariant P0: $s < i$.

We first deal with that precondition $m < i$ of assignment *s := i-1*. Global correctness is for free (Widening); local correctness follows directly from preceding assignment *m := i-1*.

We are left with invariant P0: $s < i$. For initial correctness we adopt P0 as precondition. Maintenance is for free (Widening).

Thus we obtain:

Pre: i = 0 ∧ P0	
W: **priv**: i, x * [WRITE(x.i) ; i := i+1]	R: **loc**: m, s * [m := i-1 ; {m < i} s := i-1 ; {m ≤ s} READ(x.s)]
Inv: P0: s < i	
Con: C0: *s* is ascending in time	

Note that this solution contains no queried items. Variable *m* is never inspected, because in fact variable *s* imitates variable *m*. Therefore we eliminate variable *m*:

Pre: i = 0 ∧ P0	
W: **priv**: i, x * [WRITE(x.i) ; i := i+1]	R: **loc**: s * [s := i-1 ; READ(x.s)]
Inv: P0: s < i	
Con: C0: *s* is ascending in time	

Solution 0

Note that this solution only contains one-point atomic statements (Private Occurrences).

9.2 Reducing space complexity

Solution 0 assumes in some sense that entire array *x* is available, which size is unbounded. We want to reduce the number of elements from array *x* that we assume available. Therefore we introduce a fresh private array *y* in component W that contains some elements of *x* such that array *x* can be eliminated. In this section we investigate which elements we need.

The only statement that reads an element from *x* is *READ(x.s)* in component R. Because array *x* will be eliminated, we want to replace statement *READ(x.s)* by statement *READ(y.t)* with *t* a fresh local variable of component R. Then we have to require precondition *x.s = y.t* of *READ(y.t)*.

We now deal with that precondition *x.s = y.t* of *READ(y.t)* in component R. Global correctness follows from invariant P0 (*s < i*) and hence *s ≠ i*. For local correctness we extend the preceding assignment *s := i-1* with assignment *t := h*, with *h* a fresh private variable of component W. We adopt required precondition *x.(i-1) = y.h* in component R as invariant P1, because it is only about private variables of component W.

Pre: i = 0 ∧ P0	
W: **priv**: h, i, x, y * [WRITE(x.i) ; i := i+1]	R: **loc**: s, t * [s, t := i-1, h ; {x.s = y.t} READ(y.t)]
Inv: P0: s < i P1: **?** x.(i-1) = y.h	
Con: C0: *s* is ascending in time	

We are left with invariant P1: *x.(i-1) = y.h*. For initial correctness we adopt P1 as precondition. For maintenance we extend assignment *i := i+1* with assignment *h := j*, with *j* a fresh local variable of component W. We require precondition *x.i = y.j*.

We now deal with that precondition *x.i = y.j* of assignment *i, h := i+1, j* in component W. Global correctness is for free (Private Variables). For local correctness we extend the single *WRITE(x.i)* into the combined *WRITE(x.i, y.j)* that writes the same value in both *x.i* and *y.j*. This statement can violate invariant P1 (*x.(i-1) = y.h*) and assertion *x.s = y.t* in component R. For the invariant we have require precondition *j ≠ h*; for the assertion we have to require *j ≠ t* as precondition due to invariant P0(*s < i*) which implies *s ≠ i*.

Pre: i = 0 ∧ P0 ∧ P1	
W: **loc**: j **priv**: h, i, x, y * [{**?** j ≠ h}{**?** j ≠ t} WRITE(x.i, y.j) ; {x.i = y.j} i, h := i+1, j]	R: **loc**: s, t * [s, t := i-1, h ; {x.s = y.t} READ(y.t)]
Inv: P0: s < i P1: x.(i-1) = y.h	
Con: C0: *s* is ascending in time	

We first deal with precondition *j ≠ h* of *WRITE(x.i, y.j)* in component W. Global correctness is for free (Private Variables). For local correctness we insert an assignment *j: j ≠ h*.

We are left with precondition *j ≠ t* of *WRITE(x.i, y.j)* in component W. Global correctness is guaranteed by co-assertion *j ≠ h*. For local correctness we make variable *t* a private variable and strengthen assignment *j: j ≠ h* into assignment *j: j ≠ h ∧ j ≠ t*.

Thus we obtain:

Pre: i = 0 ∧ P0 ∧ P1	
W: **loc**: j **priv**: h, i, x, y * [j: j ≠ h ∧ j ≠ t ; {j ≠ h}{j ≠ t} WRITE(x.i, y.j) ; {x.i = y.j} i, h := i+1, j]	R: **loc**: s **priv**: t * [s, t := i-1, h ; {x.s = y.t} READ(y.t)]
Inv: P0: s < i P1: x.(i-1) = y.h	
Con: C0: *s* is ascending in time	

We now eliminate variables *x*, *s* and *i*:

W: **loc**: j **priv**: h, y * [j: j ≠ h ∧ j ≠ t ; {j ≠ h}{j ≠ t} WRITE(y.j) ; h := j]	R: **priv**: t * [t := h ; READ(y.t)]

Solution 1

Also note that this solution assumes that variable *j* is (at least) three-valued and so are variables *h* and *t*. So we can implement this algorithm by using an array *y* with only three elements.

9.3 Towards two-valued variables

It turns out to be hard to find an assignment with a compact representation that implements the assignment to *j*, because it has to deal with two assertions: $j \neq h$ and $j \neq t$. To create more manipulative freedom, we split variables *j*, *h* and *t* into two-tuples:

j := (p, q) h := (f, g) t := (u, v)

To keep the notation clear, we write tuple assignments as multiple assignments and arrays with tuple indices as matrices.

W: **loc**: p, q **priv**: f, g, y * [p, q: (p, q) ≠ (f, g) ∧ (p, q) ≠ (u, v) ; {(p, q) ≠ (f, g)}{(p, q) ≠ (u, v)} WRITE(y.p.q) ; f, g := p, q]	R: **priv**: u, v * [u, v := f, g ; READ(y.u.v)]

We now deal with the two conjuncts of assignment *p, q: (p, q) ≠ (f, g) ∧ (p, q) ≠ (u, v)* in component W separately:

```
        (p, q) ≠ (f, g)
≡           { calculus, eliminate tuple notation }
        p ≠ f  ∨  q ≠ g
⇐           { calculus, breaking symmetry }
        q ≠ g
```

We use this condition in the assignment to *p* and *q*. Because variables *g* and *q* are private variables of component W, we also replace assertion *(p, q) ≠ (f, g)* by assertion *q ≠ g*.

```
        (p, q) ≠ (u, v)
≡           { calculus, eliminate tuple notation }
        p ≠ u  ∨  q ≠ v
⇐           { calculus, breaking symmetry by eliminating variable q }
        p ≠ u
```

We use this condition in the assignment to *p* and *q*. Then variable *v* becomes a local variable.

Thus we obtain:

```
W: loc:   p, q
   priv: f, g, y
   * [ p, q : p ≠ u  ∧ q ≠ g
     ; {q ≠ g}{p ≠ u  ∨  q ≠ v}
       WRITE( y.p.q )
     ; f, g := p, q
     ]
```

```
R: loc:  v
   priv: u
   * [ u, v := f, g
     ; READ( y.u.v )
     ]
```

Solution 2

Note that this solution assumes that *p* and *q* are (at least) two-valued and so are *f*, *g*, *u* and *v*. It is now easy to implement the assignment to *p* and *q*, but (at least) four elements in matrix *y* are necessary.

Now we want to find out which assignment to *j* in solution 1 would correspond to solution 2. For this moment we assume that *j* is a two-bit variable and we consider *p* as the most significant and *q* as the least significant bit of *j*, and similarly for variables *h* and *t*. We map bits on the integers *0* and *1*. Then we calculate for the assignment to *j*:

```
        j
≡           { representation of j:          j = 2 * p + q }
        2 * p + q
≡           { assignment to p and q:        p, q := 1- u, 1- g }
        2 * (1- u) + (1- g)
≡           { representation of t and h:    u = t div 2  and  g = h mod 2 }
        2 * (1- t div 2) + (1- h mod 2)
```

Thus the one-point (Private Occurrences) assignment to *j* in solution 1 that corresponds with solution 2 is *j := 2 * (1- t **div** 2) + (1- h **mod** 2)*. Using this assignment, variable *j* is four-valued, and so are variables *h* and *t*. Thus four elements in array *y* are necessary.

9.4 Implementation

We now continue with solution 2. We want to implement the multiple assignment to *p* and *q*. We decide to implement variables *p*, *q*, and so *f*, *g*, *u* and *v*, as two-valued variables. Then that assignment becomes *p, q := ¬u, ¬g*.

Observe that every inspection of variable *q* has precondition $q \neq g$ (Private Variables). Therefore we eliminate variable *q* and everywhere replace variable *q* by *¬g*.

Thus we obtain:

```
W: loc:  p                          R: loc:  v
   priv: f, g, y                       priv: u
   * [ p := ¬u                         * [ u, v := f, g
     ; {p ≠ u  ∨  g = v}                 ; READ( y.u.v )
       WRITE( y.p.(¬g) )                 ]
     ; f, g := p, ¬g
     ]
```

Solution 3

Although this algorithm looks somewhat like the one in [Hes98], it is more coarse-grained. To obtain a fine-grained solution we have to make both multiple assignments one-point atomic statements. Because this (especially for assignment *u, v := f, g*) turns out to be hard, we decide not to continue this derivation.

10. Fine-grained wait-free handshake register

This chapter describes a derivation of a fine-grained algorithm for a wait-free handshake register and the implementation as described in [Hes98].

10.0 Specification

A handshake register controls access to a data structure shared by a writer and a reader. The writer repeatedly produces information and writes it into the data structure. The reader repeatedly reads information from the data structure and outputs it. Writing into the data structure should not overlap with some other writing into or reading from the data structure.

One may wonder whether non-overlap and wait-freeness are conflicting requirements. If we are prepared to introduce multiple copies of the shared data structure, they do not have to conflict. For instance if we ensure that writing and reading occurs in different copies.

To develop a formal specification, we consider the following two-component multiprogram:

Pre: j = 0	
W: **loc**: j, s, x * [x.j, j := s.x, j+1]	R: **loc**: y * [print(y)]

Computation Proper

Component W produces information by filling local array *x*. Local variable *j* indicates the number of terminated production phases, so that *x[0..j)* is produced. Local function *s* is just an abstraction of the production of a next value in array *x*. Component R outputs information by printing the value of local variable *y*.

We now introduce some terminology. The computation proper describes the entire production phase of component W and the entire output phase of component R. The production phase of component W will be followed by a write phase that writes the last produced element in a shared data structure. The output phase of component R will be preceded by a read phase that reads a value from a shared data structure.

Now we can specify how these components have to be synchronised. We consider three (functional) requirements and three (non-functional) constraints.

The first requirement we consider is that the values printed by component R are produced values from array *x*. Therefore we introduce fresh local variable *z* in component R. Then element *y* is an element from *x*, if we require that at the end of the read phase variable *z* indicates the index of an element *y* from *x*. Element *y* is produced, if we also require that *z* is less than *j*.

Pre: j = 0	
W: **loc**: j, s, x * [x.j, j := s.x, j+1 …]	R: **loc**: y, z * [… {**?** y = x.z}{**?** z < j} print(y)]

The next requirement we consider is recentness. Recentness means that the value of *y* at the end of the read phase was at some time during that read phase the most recently written value. To specify this, we introduce fresh private variable *J* in component W to record the number of terminated write phases, that is the number of elements of *x* written in the shared data structure. We introduce fresh local variable *m* in component R to store at the beginning of the read phase the index of the element of *x* that was most recently written in the shared data structure as indicated by variable *J*. We require that at the end of the read phase *z* is at least *m*.

One may wonder whether component W could be expressed without variable *j*, because each inspection of variable *j* has precondition *j = J*. We use both variables *j* and *J* to be able to express the difference between what is produced and what is completely written.

Pre: j = 0 ∧ J = 0	
W: **loc**: j, s, x **priv**: J * [x.j, j := s.x, j+1 … ; J := J+1]	R: **loc**: m, y, z * [m := J-1 … ; {**?** y = x.z}{**?** z < j}{**?** m ≤ z} print(y)]

The last requirement we consider is sequentiality. Sequentiality means that for two read phases, the value read during the later phase is not older than the value read during the previous phase. Therefore we require as constraint C0 that the value of z is ascending in time.

Pre: j = 0 ∧ J = 0	
W: **loc**: j, s, x **priv**: J * [x.j, j := s.x, j+1 … ; J := J+1]	R: **loc**: m, y, z * [m := J-1 … ; {**?** y = x.z}{**?** z < j}{**?** m ≤ z} print(y)]
Con: C0: **?** z is ascending in time	

Specification

The problem thus specified has to be solved under three extra constraints, to wit

- that the final algorithm be entirely expressed in one-point atomic statements;
- that the final algorithm contain no write operation that overlaps some other write or read operation on the same shared data structure;
- that the final algorithm be wait-free (see chapter 4).

10.1 Component R prints produced values from x

We first deal with assertion $y = x.z$ in component R. Thanks to co-assertion $z < j$ and hence $z \neq j$, global correctness of assertion $y = x.z$ in component R is guaranteed. Because array x is a local variable of component W and variables y and z are local variables of component R, local correctness can only be established by communication (via private or shared variables). It is not desirable to communicate entire array x due to its unlimited size, so we will communicate values for variables y and z. We will now investigate how to do this.

For local correctness of assertion $y = x.z$ we have to introduce assignments to y and z. Assume for this moment that E and F are fresh shared variables. Then consider the following three constructions to ensure local correctness of assertion $y = x.z$:

```
 {? E = x.F}          {? E = x.F}          {? E = x.F}
 y := E               z := F               y, z := E, F
;{? y = x.F}         ;{? E = x.z}         {? y = x.z}
 z := F               y := E
 {? y = x.z}          {? y = x.z}
```

Although the left two constructions contain only one-point statements, we will use the right-most construction because it has a weaker proof obligation.

In the previous section, we concluded that storing one copy of the data-structure would not be sufficient in the final solution (due to non-overlap and wait-freeness). Therefore we directly introduce *arrays* Y and Z to store some solutions of $y, z: y = x.z$. Because array x is a local variable of component W, we introduce arrays Y and Z as private variables of component W. The non-overlapping constraint is applicable to the elements of array Y, so we must instantiate this constraint for array Y as constraint C1:

C1: “no write operation overlaps some operation on the same element of Y” .

Note that we could have introduced one array of two-tuples, but this would make the notation in the rest of this derivation more complicated, because array Z will be used much more than array Y.

We introduce local variable b in component R to indicate an index in both Y and Z. For local correctness of assertion $y = x.z$ in component R, we introduce an assignment $y, z := Y.b, Z.b$ for which we calculate a precondition:

```
        (y, z := Y.b, Z.b).(y = x.z)
≡           { substitution }
        Y.b = x.(Z.b)
⇐           { generalize b, it turns out to simplify the derivation (number of assertions) }
        ( ∀v :: Y.v = x.(Z.v) )
```

We adopt this condition as invariant P0.

Pre: j = 0 ∧ J = 0	
W: **loc**: j, s, x **priv**: J, Y, Z * [x.j, j := s.x, j+1 … ; J := J+1]	R: **loc**: b, m, y, z * [m := J-1 … ; y, z := Y.b, Z.b ; {y = x.z}{**?** z < j}{**?** m ≤ z} print(y)]
Inv: P0: **?** (∀v :: Y.v = x.(Z.v))	
Con: C0: **?** *z* is ascending in time C1: **?** no write operation overlaps some operation on the same element of *Y*	

We first deal with assertion $z < j$ in component R. Global correctness is for free (Widening). For local correctness we calculate:

(y, z := Y.b, Z.b).(z < j)
≡ { substitution }
Z.b < j
⇐ { generalize *b*, it turns out to simplify the derivation (see for example P0) }
(∀v :: Z.v < j)

We adopt this condition as invariant P1.

We now deal with invariant P0: *(∀v :: Y.v = x.(Z.v))*. For initial correctness we adopt P0 as precondition. Maintenance is guaranteed because, by invariant P1, *(∀v :: Z.v ≠ j)*.

We now deal with invariant P1: *(∀v :: Z.v < j)*. For initial correctness we adopt P1 as precondition. Maintenance is for free (Widening).

Thus we obtain:

Pre: j = 0 ∧ J = 0 ∧ P0 ∧ P1	
W: **loc**: j, s, x **priv**: J, Y, Z * [x.j, j := s.x, j+1 … ; J := J+1]	R: **loc**: b, m, y, z * [m := J-1 … ; y, z := Y.b, Z.b ; {y = x.z}{z < j}{**?** m ≤ z} print(y)]
Inv: P0: (∀v :: Y.v = x.(Z.v)) P1: (∀v :: Z.v < j)	
Con: C0: **?** *z* is ascending in time C1: **?** no write operation overlaps some operation on the same element of *Y*	

10.2 Recentness

We first deal with assertion $m \leq z$ in component R. Global correctness is for free (Private Variables). Local correctness follows from the preceding assignment *y, z := Y.b, Z.b* for which we require precondition $m \leq Z.b$.

We then deal with that precondition $m \leq Z.b$ of assignment *y, z := Y.b, Z.b* in component R. Global correctness will be guaranteed in what follows. Because variable *Z* is a private variable of component W, we have to introduce an invariant to establish local correctness, but we will first reduce the number of local variables of component R that are involved in such condition. Because component R already contains assignment *m := J-1*, we will eliminate variable *b* from such invariant. We introduce a fresh private variable *A* in component W and insert assignment *b := A* in component R, for which we calculate a precondition:

$$
\begin{array}{ll}
 & (b := A).(m \leq Z.b) \\
\equiv & \quad \{ \text{ substitution } \} \\
 & m \leq Z.A \\
\Leftarrow & \quad \{ \text{ transitivity } \leq \text{, obtain conjunct } m < J \text{ which is a candidate precondition by preceding assignment } m := J\text{-}1 \ \} \\
 & m < J \ \wedge \ J\text{-}1 \leq Z.A
\end{array}
$$

We adopt the first conjunct as precondition of assignment *b := A* and the second conjunct as invariant P2.

We now deal with precondition $m < J$ of assignment *b := A* in component R. Local correctness follows directly from preceding assignment *m := J-1*. Global correctness is for free (Widening).

Pre:	$j = 0 \ \wedge \ J = 0 \ \wedge$ P0 $\wedge$ P1	
W: **loc**: j, s, x **priv**: A, J, Y, Z * [x.j, j := s.x, j+1 … ; J := J+1]		R: **loc**: b, m, y, z * [m := J-1 ; {m < J} b := A ; {m ≤ Z.b} y, z := Y.b, Z.b ; {y = x.z}{z < j}{m ≤ z} print(y)]
Inv:	P0:	$(\forall v :: Y.v = x.(Z.v))$
	P1:	$(\forall v :: Z.v < j)$
	P2:	**?** J-1 ≤ Z.A
Con:	C0:	**?** *z* is ascending in time
	C1:	**?** no write operation overlaps some operation on the same element of *Y*

We first deal with invariant P2: *J-1 ≤ Z.A*. For initial correctness we adopt P2 as precondition. For maintenance we require precondition *J ≤ Z.A* of assignment *J := J+1* in component W.

We then deal with that precondition $J \leq Z.A$ of assignment $J := J+1$ in component W. Global correctness is for free (Private Variables). For local correctness we could directly introduce an assignment to $Z.A$, that must be accompanied by an assignment to Y (by invariant P0). But then it will be hard to fulfil constraint C1, because of assignment $b := A$ before inspecting Z. Therefore we will first reduce the number of private variables that are involved in this assertion. We introduce local variable a in component W and insert assignment $A := a$ with precondition $J \leq Z.a$:

```
{? J ≤ Z.a}
A := a
{J ≤ Z.A}
```

Thanks to precondition $J \leq Z.a$ of assignment $A := a$ and hence $J-1 \leq Z.a$, assignment $A := a$ maintains invariant P2.

We then deal with that precondition $J \leq Z.a$ of assignment $A := a$ in component W. Global correctness is for free (Private Variables). For local correctness we introduce an assignment $Z.a := J$. This assignment can endanger invariants P0 and P1 and assertion $m \leq Z.b$ in component R.

For maintenance of invariant P0, we have to extend assignment $Z.a := J$ with an assignment $Y.a := x.J$. Then we must also deal with constraint C1. We do so by introducing precondition P of assignment $Y.a, Z.a := x.J, J$ in component W, precondition Q of assignment $y, z := Y.b, Z.b$ in component R and proof obligation $P \wedge Q \Rightarrow a \neq b$. Because assertion Q is a co-assertion of assertion $m \leq Z.b$, global correctness of the latter assertion under {? P} $Y.a, Z.a := x.J, J$ is guaranteed (Orthogonality). Maintenance of invariant P1 follows from precondition $J < j$ (by topology of component W).

Thus we obtain:

<table>
<tr><td colspan="2">Pre: j = 0 ∧ J = 0 ∧ P0 ∧ P1 ∧ P2</td></tr>
<tr><td>W: loc: a, j, s, x
priv: A, J, Y, Z
* [x.j, j := s.x, j+1
…
; {? P}
Y.a, Z.a := x.J, J
; {J ≤ Z.a}
A := a
; {J ≤ Z.A}
J := J+1
]</td><td>R: loc: b, m, y, z
* [m := J-1
; {m < J}
b := A
; {m ≤ Z.b}{? Q}
y, z := Y.b, Z.b
; {y = x.z}{z < j}{m ≤ z}
print(y)
]</td></tr>
<tr><td colspan="2">Inv: P0: (∀v :: Y.v = x.(Z.v))
P1: (∀v :: Z.v < j)
P2: J-1 ≤ Z.A</td></tr>
<tr><td colspan="2">Prf: ? P ∧ Q ⇒ a ≠ b</td></tr>
<tr><td colspan="2">Con: C0: ? z is ascending in time
C1: no write operation overlaps some operation on the same element of Y</td></tr>
</table>

10.3 Sequentiality

We first deal with constraint C0: *z* is ascending in time. For maintenance we require $z \leq Z.b$ as precondition of assignment *y, z := Y.b, Z.b* in component R.

We then deal with that precondition $z \leq Z.b$ of assignment *y, z := Y.b, Z.b* in component R. Thanks to co-assertion *Q* of assertion $z \leq Z.b$, global correctness of this assertion under {P} *Y.a, Z.a := x.J, J* is guaranteed (Orthogonality). Local correctness follows from preceding assignment *b := A* if we require invariant P3: $z \leq Z.A$.

We then deal with that invariant P3: $z \leq Z.A$. For initial correctness we adopt P3 as precondition. Maintenance can be endangered by three assignments for which we calculate:

|[$J \leq Z.a$
▷ (A := a).P3
≡ { substitution }
$z \leq Z.a$
⇐ { precondition $J \leq Z.a$ }
$z \leq J$
]|

We adopt this condition as invariant P4.

(Y.a, Z.a := x.J, J).P3
≡ { case analysis; substitution; target invariant P3 }
$a = A \Rightarrow z \leq J$
≡ { invariant P4: $z \leq J$ }
true

(y, z := Y.b, Z.b).P3
≡ { substitution }
$Z.b \leq Z.A$

We adopt this condition as precondition of assignment *y, z := Y.b, Z.b* in component R. Note that this condition would follow from invariant P1 and P2 ($Z.b \leq j\text{-}1 \wedge J\text{-}1 \leq Z.A$) if we require $j \leq J$. Unfortunately, this condition is hopeless both as assertion in component R and as system invariant.

We now deal with invariant P4: $z \leq J$. For initial correctness we adopt P4 as precondition. Assignment *J := J+1* in component W maintains invariant P4 (Widening). Maintenance can be endangered by assignment *y, z := Y.b, Z.b* for which we calculate:

(y, z := Y.b, Z.b).P4
≡ { substitution }
$Z.b \leq J$
≡ { invariant P1 with *v := b*; topology of component W: $j\text{-}1 \leq J$ }
true

We now deal with precondition $Z.b \le Z.A$ of assignment $y, z := Y.b, Z.b$ in component R. Local correctness follows from the preceding assignment $b := A$. Global correctness can be endangered by two assignments for which we calculate:

```
|[  Z.b ≤ Z.A  ∧  Q  ∧  P
▷          (Y.a, Z.a := x.J, J).(Z.b ≤ Z.A)
   ≡              { case analysis; substitution; target assertion Z.b ≤ Z.A }
          (  a = b  ∧  a = A   ⇒   J ≤ J  )
        ∧ (  a = b  ∧  a ≠ A   ⇒   J ≤ Z.A  )
        ∧ (  a ≠ b  ∧  a = A   ⇒   Z.b ≤ J  )
   ≡              { a ≠ b, use precondition P and co-assertion Q }
          a = A  ⇒  Z.b ≤ J
   ≡              { invariant P1 with v := b; topology of component W: j-1 ≤ J }
          true
]|

|[  J ≤ Z.a
▷          (A := a).(Z.b ≤ Z.A)
   ≡              { substitution }
          Z.b ≤ Z.a
   ⇐              { precondition J ≤ Z.a }
          Z.b ≤ J
   ≡              { invariant P1 with v := b; topology of component W: j-1 ≤ J }
          true
]|
```

Thus we obtain:

<table>
<tr><td colspan="2">Pre: j = 0 ∧ J = 0 ∧ P0 ∧ P1 ∧ P2 ∧ P3 ∧ P4</td></tr>
<tr><td>W: loc: a, j, s, x
priv: A, J, Y, Z
* [x.j, j := s.x, j+1
…
; {? P}
Y.a, Z.a := x.J, J
; {J ≤ Z.a}
A := a
; {J ≤ Z.A}
J := J+1
]</td><td>R: loc: b, m, y, z
* [m := J-1
; {m < J}
b := A
; {m ≤ Z.b}{? Q}{z ≤ Z.b}{Z.b ≤ Z.A}
y, z := Y.b, Z.b
; {y = x.z}{z < j}{m ≤ z}
print(y)
]</td></tr>
<tr><td colspan="2">Inv: P0: (∀v :: Y.v = x.(Z.v))
P1: (∀v :: Z.v < j)
P2: J-1 ≤ Z.A
P3: z ≤ Z.A
P4: z ≤ J</td></tr>
<tr><td colspan="2">Prf: ? P ∧ Q ⇒ a ≠ b</td></tr>
<tr><td colspan="2">Con: C0: z is ascending in time
C1: no write operation overlaps some operation on the same element of Y</td></tr>
</table>

10.4 Non-overlap

Observe that there are no shared variables and component R contains no private variables. This means that all communication is uni-directional from component W to component R. If we assume that we can only implement arrays with a bounded domain, component W must be able to write elements in *Y* more than once. Although component R must read elements in *Y* written by component W, writing and reading may not overlap. Therefore communication from component R to component W is necessary. So we must be prepared to introduce private variable(s) in component R.

We are left with proof obligation $P \wedge Q \Rightarrow a \neq b$. Observe that variables *a* and *b* are local variables of component W and R respectively, so we have to allow for communication (via private or shared variables). Therefore we calculate:

|[$P \wedge Q$
▷ $a \neq b$
⇐ { introduce fresh private variable *B* in component R. Enable communication from component R to component W via variable *B*. }
$a \neq B \wedge B = b$
⇐ { introduce *P'* and *Q'* with proof obligation $P' \wedge Q' \Rightarrow a \neq B$. We first want to deal with conjunct $B = b$ containing variables of one component only }
$P' \wedge Q' \wedge B = b$
≡ { choose $P \equiv P'$ and $Q \equiv Q' \wedge B = b$, because variables *B* and *b* are variables of component R }
$P \wedge Q$
]|

We now deal with co-assertion $B = b$ of assertion *Q'* in component R. Global correctness is for free (Private Variables). For local correctness we extend preceding assignment $b := A$ with assignment $B := A$.

We are left with proof obligation $P' \wedge Q' \Rightarrow a \neq B$. We will use the approach described in chapter 3, that is we will prove *only local* correctness of such assertion $a \neq B$ as co-assertion of *both P'* and *Q'*. Observe that no co-assertion of *P'* in component W contains variable *a* so we can safely insert an assignment to variable *a* preceding *P'*. This means that component W can easily establish local correctness of assertion $a \neq B$ by adding an assignment to variable *a*. However, component R can not easily establish local correctness of $a \neq B$ by adding an assignment to variable *B*, because variable *a* is a local variable of component W and by co-assertion $B = b$ of Q'.

Before we continue we want to create more manipulative freedom. Therefore we split variables into two-tuples:

a := (c, d) b := (e, f) A := (C, D) B := (E, F)

To keep the notation clear, we write tuple assignments as multiple assignments and arrays with tuple indices as matrices.

Pre: j = 0 ∧ J = 0 ∧ P0 ∧ P1 ∧ P2 ∧ P3 ∧ P4	
W: **loc**: c, d, j, s, x **priv**: C, D, J, Y, Z * [x.j, j := s.x, j+1 … ; {**?** P'} Y.c.d, Z.c.d := x.J, J ; {J ≤ Z.c.d } C, D := c, d ; {J ≤ Z.C.D} J := J+1]	R: **loc**: e, f, m, y, z **priv**: E, F * [m := J-1 ; {m < J} e, f, E, F := C, D, C, D ; {m ≤ Z.e.f}{**?** Q'}{E = e}{F = f}{z ≤ Z.e.f} {Z.e.f ≤ Z.C.D} y, z := Y.e.f, Z.e.f ; {y = x.z}{z < j}{m ≤ z} print(y)]
Inv: P0: (∀v, w :: Y.v.w = x.(Z.v.w)) P4: z ≤ J P1: (∀v, w :: Z.v.w < j) P2: J-1 ≤ Z.C.D P3: z ≤ Z.C.D	
Prf: **?** P' ∧ Q' ⇒ (c, d) ≠ (E, F)	
Con: C0: *z* is ascending in time C1: no write operation overlaps some operation on the same element of *Y*	

We now calculate for $P' \wedge Q' \Rightarrow (c, d) \neq (E, F)$:

|[P' ∧ Q'

▷ (c, d) ≠ (E, F)

≡ { eliminate tuple notation }

c ≠ E ∨ d ≠ F

≡ { calculus, because variables *c* and *d* are local variables of component W and variables *E* and *F* are private variables of component R, component W component can easily establish local correctness of both disjuncts, but component R of none of the disjuncts. We will manipulate one of the disjuncts such that component R can establish (its) local correctness }

c = E ⇒ d ≠ F

⇐ { introduce fresh private array *G* in component W, use antecedent *c = E*. Separate local variable *d* of component W and private variable *F* of component R. }

c = E ⇒ (d ≠ G.c ∧ G.E = F)

≡ { calculus; distribute ∨ over ∧ }

(c ≠ E ∨ d ≠ G.c) ∧ (c ≠ E ∨ G.E = F)

⇐ { introduce *P''* and *Q''* with proof obligation $P'' \wedge Q'' \Rightarrow c \neq E \vee G.E = F$, because we first want to deal with term $c \neq E \vee d \neq G.c$ }

(c ≠ E ∨ d ≠ G.c) ∧ P'' ∧ Q"

≡ { choose $P' \equiv (c \neq E \vee d \neq G.c) \wedge P''$ and $Q' \equiv Q''$ }

P' ∧ Q'

]|

We now deal with co-assertion $c \neq E \ \vee \ d \neq G.c$ of P'' in component W. For global correctness we strengthen it into assertion $d \neq G.c$ (Private Variables). For local correctness of assertion $d \neq G.c$ we insert an assignment $d: d \neq G.c$.

We now deal with proof obligation $P'' \wedge Q'' \Rightarrow c \neq E \ \vee \ G.E = F$ as described in chapter 3. For local correctness of (the first disjunct of) the consequent in component W we extend the assignment to d with an assignment $c: c \neq E$. For local correctness of (the second disjunct of) the consequent in component R we have to use assignment $e, f, E, F := C, D, C, D$ for which we calculate a precondition:

```
        (e, f, E, F := C, D, C, D).(G.E = F)
≡           { substitution }
        G.C = D
```

Because this condition contains only variables of component W, we adopt it as invariant P5.

We are left with invariant P5: $G.C = D$. For initial correctness we adopt P5 as precondition. For maintenance we extend assignment $C, D := c, d$ with assignment $G.c := d$.

Thus we obtain:

<table>
<tr><td colspan="2">Pre: j = 0 ∧ J = 0 ∧ P0 ∧ P1 ∧ P2 ∧ P3 ∧ P4 ∧ P5</td></tr>
<tr><td>W: loc: c, d, j, s, x
priv: C, D, G, J, Y, Z
* [x.j, j := s.x, j+1
; <c, d : c ≠ E ∧ d ≠ G.c {c ≠ E}>
; {d ≠ G.c}
Y.c.d, Z.c.d := x.J, J
; {J ≤ Z.c.d}
C, D, G.c := c, d, d
; {J ≤ Z.C.D}
J := J+1
]</td><td>R: loc: e, f, m, y, z
priv: E, F
* [m := J-1
; {m < J}
<e, f, E, F := C, D, C, D {G.E = F}>
; {m ≤ Z.e.f}{E = e}{F = f}{z ≤ Z.e.f}
{Z.e.f ≤ Z.C.D}
y, z := Y.e.f, Z.e.f
; {y = x.z}{z < j}{m ≤ z}
print(y)
]</td></tr>
<tr><td colspan="2">Inv: P0: (∀v, w :: Y.v.w = x.(Z.v.w)) P4: z ≤ J
P1: (∀v, w :: Z.v.w < j) P5: G.C = D
P2: J-1 ≤ Z.C.D
P3: z ≤ Z.C.D</td></tr>
<tr><td colspan="2">Con: C0: z is ascending in time
C1: no write operation overlaps some operation on the same element of Y</td></tr>
</table>

Note that this solution contains no queried items.

10.5 Grain-size

Before we make this solution fine-grained, we first eliminate some redundant variables.

Observe that private variable *F* of component R is not used in component W so it is in fact a local variable of component R. We could maintain *F = f* as invariant P6 by initially requiring P6. Invariant P6 means that variable *F* is an alias of *f*. Thus we can safely eliminate variable *F* and everywhere replace variable *F* by *f*.

Observe that invariant P5 (*G.C = D*) means that variable *D* is an alias of *G.C*. Thus we can safely eliminate variable *D* and everywhere replace variable *D* by *G.C*.

<table>
<tr><td colspan="2">Pre: j = 0 ∧ J = 0 ∧ P0 ∧ P1 ∧ P2 ∧ P3 ∧ P4</td></tr>
<tr><td>W: loc: c, d, j, s, x
priv: C, G, J, Y, Z
* [x.j, j := s.x, j+1
; <c, d : c ≠ E ∧ d ≠ G.c {c ≠ E}>
; {d ≠ G.c}
Y.c.d, Z.c.d := x.J, J
; {J ≤ Z.c.d}
C, G.c := c, d
; {J ≤ Z.C.(G.C)}
J := J+1
]</td><td>R: loc: e, f, m, y, z
priv: E
* [m := J-1
; {m < J}
<e, f, E := C, G.C, C {G.E = f}>
; {m ≤ Z.e.f}{E = e}{z ≤ Z.e.f}
{Z.e.f ≤ Z.C.(G.C)}
y, z := Y.e.f, Z.e.f
; {y = x.z}{z < j}{m ≤ z}
print(y)
]</td></tr>
<tr><td colspan="2">Inv: P0: (∀v, w :: Y.v.w = x.(Z.v.w)) P4: z ≤ J
P1: (∀v, w :: Z.v.w < j)
P2: J-1 ≤ Z.C.(G.C)
P3: z ≤ Z.C.(G.C)</td></tr>
<tr><td colspan="2">Con: C0: z is ascending in time
C1: no write operation overlaps some operation on the same element of Y</td></tr>
</table>

This solution contains six non-one-point assignments that must finally be eliminated:

c, d : c ≠ E ∧ d ≠ G.c	e, f, E := C, G.C, C
Y.c.d, Z.c.d := x.J, J	y, z := Y.e.f, Z.e.f
C, G.c := c, d	
J := J+1	

Re: *Y.c.d, Z.c.d := x.J, J* and *y, z := Y.e.f, Z.e.f*
Because variable *z* is not inspected in this solution, we will finally eliminate it and then variable *Z* will also be eliminated. Then these two assignments become single one-point statements.

Re: *c, d : c ≠ E ∧ d ≠ G.c* and *J := J+1*
These two assignments are in fact one-point statements (Private Occurrences).

For the other two assignments, our strategy will be to rewrite a (non-one-point) assignment into a couple of (one-point) assignments that maintain local correctness of the post-assertion of that assignment. Then we must prove global correctness of new intermediate assertions and we must prove that the invariants and the assertions in other components can not be endangered by these new assignments.

For example, consider a multiple assignment

{U} m, n := M, N {W}

We try to find *M'*, *N'* and *V* such that local correctness of *V* and *W* is guaranteed in

{U} m := M' ;{**?** V} n := N' {W}

Then we must prove global correctness of assertion *V* and we must prove that the invariants and the assertions in other components can not be endangered by assignments {U} *m := M'* and {V} *n := N'*.

Re: {J ≤ Z.c.d} *C, G.c := c, d* {J ≤ Z.C.(G.C)}

To make this assignment fine-grained, we split it into its single one-point assignments:

{J ≤ Z.c.d}
G.c := d
; {**?** J ≤ Z.c.(G.c)}
C := c
{J ≤ Z.C.(G.C)}

We now consider new intermediate assertion *J ≤ Z.c.(G.c)*. Local correctness is guaranteed by construction. Global correctness is for free (Private Variables). The two new assignments can endanger global correctness of assertion *Z.e.f ≤ Z.C.(G.C)* in component R and maintenance of invariant P2 (*J-1 ≤ Z.C.(G.C)*) and P3 (*z ≤ Z.C.(G.C)*). Therefore we calculate:

|[Z.e.f ≤ Z.C.(G.C) ∧ J ≤ Z.c.d
▷ (G.c := d).(Z.e.f ≤ Z.C.(G.C))
≡ { case analysis; substitution; target assertion *Z.e.f ≤ Z.C.(G.C)* }
c = C ⇒ Z.e.f ≤ Z.c.d
⇐ { precondition *J ≤ Z.c.d* }
c = C ⇒ Z.e.f ≤ J
≡ { invariant P1 with *v, w := e, f*; topology of component W: *j-1 ≤ J* }
true
]|

|[J ≤ Z.c.(G.c)
▷ (C := c).(Z.e.f ≤ Z.C.(G.C))
≡ { substitution }
Z.e.f ≤ Z.c.(G.c)
⇐ { precondition *J ≤ Z.c.(G.c)* }
Z.e.f ≤ J
≡ { invariant P1 with *v, w := e, f*; topology of component W: *j-1 ≤ J* }
true
]|

```
|[  J ≤ Z.c.d
▷           (G.c := d).P2
   ≡              { case analysis; substitution; target invariant P2 }
           c = C  ⇒  J-1 ≤ Z.c.d
   ≡              { precondition J ≤ Z.c.d }
           true
]|

|[  J ≤ Z.c.(G.c)
▷           (C := c).P2
   ≡              { substitution}
           J-1 ≤ Z.c.(G.c)
   ≡              { precondition J ≤ Z.c.(G.c) }
           true
]|

|[  J ≤ Z.c.d
▷           (G.c := d).P3
   ≡              { case analysis; substitution; target invariant P3 }
           c = C  ⇒  z ≤ Z.c.d
   ⇐              { precondition J ≤ Z.c.d }
           c = C  ⇒  z ≤ J
   ≡              { invariant P4: z ≤ J }
           true
]|

|[  J ≤ Z.c.(G.c)
▷           (C := c).P3
   ≡              { substitution}
           z ≤ Z.c.(G.c)
   ⇐              { precondition J ≤ Z.c.(G.c) }
           z ≤ J
   ≡              { invariant P4: z ≤ J }
           true
]|
```

Pre: j = 0 ∧ J = 0 ∧ P0 ∧ P1 ∧ P2 ∧ P3 ∧ P4	
W: **loc**: c, d, j, s, x **priv**: C, G, J, Y, Z * [x.j, j := s.x, j+1 ; <c, d : c ≠ E ∧ d ≠ G.c {c ≠ E}> ; {d ≠ G.c} Y.c.d, Z.c.d := x.J, J ; {J ≤ Z.c.d} G.c := d ; {J ≤ Z.c.(G.c)} C := c ; {J ≤ Z.C.(G.C)} J := J+1]	R: **loc**: e, f, m, y, z **priv**: E * [m := J-1 ; {m < J} <e, f, E := C, G.C, C {G.E = f}> ; {m ≤ Z.e.f}{E = e}{z ≤ Z.e.f} {Z.e.f ≤ Z.C.(G.C)} y, z := Y.e.f, Z.e.f ; {y = x.z}{z < j}{m ≤ z} print(y)]
Inv: P0: (∀v, w :: Y.v.w = x.(Z.v.w)) P4: z ≤ J P1: (∀v, w :: Z.v.w < j) P2: J-1 ≤ Z.C.(G.C) P3: z ≤ Z.C.(G.C)	
Con: C0: *z* is ascending in time C1: no write operation overlaps some operation on the same element of *Y*	

Re: {m < J} <*e, f, E := C, G.C, C* {G.E = f}>
{m ≤ Z.e.f}{E = e}{z ≤ Z.e.f}{Z.e.f ≤ Z.C.(G.C)}

Before we make this assignment fine-grained, note that we may strengthen the precondition with the weakest liberal precondition of the postcondition under this assignment:

{m ≤ Z.C.(G.C)}{z ≤ Z.C.(G.C)}

For reasons of clarity, we will mention these preconditions below. We split the assignment into one-point assignments. For local correctness of assertion *G.E = f*, the assignment to *f* will be the last statement. For local correctness of *E = e*, the assignment to *E* will occur after the assignment to *e*. So we construct:

{m < J}{m ≤ Z.C.(G.C)} {z ≤ Z.C.(G.C)}
e := C
; {**?** m ≤ Z.e.(G.e)}{**?** z ≤ Z.e.(G.e)}{**?** Z.e.(G.e) ≤ Z.C.(G.C)}
E := e
; {**?** m ≤ Z.e.(G.e)}{**?** E = e}{**?** z ≤ Z.e.(G.e)}{**?** Z.e.(G.e) ≤ Z.C.(G.C)}
<f := G.e {G.E = f}>
{m ≤ Z.e.f}{E = e}{z ≤ Z.e.f}{Z.e.f ≤ Z.C.(G.C)}

The three new assignments can not endanger other assertions or invariants, because variables *e*, *E* and *f* do not occur in invariants or assertions of component W that need global correctness proofs (assertion *c ≠E* does not).

We now consider the new intermediate assertions. Local correctness is guaranteed by construction. Global correctness of assertion $E = e$ is for free (Private Variables). Assignment $Y.c.d, Z.c.d := x.J, J$ in component W could possibly endanger the other assertions if $c = e \wedge d = G.e$ or if $c = C \wedge d = G.C$. Thanks to precondition $d \neq G.c$, this is prevented. So we only have to prove their global correctness under assignment $G.c := d$ in component W and only for assertion $Z.e.(G.e) \leq Z.C.(G.C)$ under assignment $C := c$ in component W:

$|[\ z \leq Z.e.(G.e) \wedge J \leq Z.c.d$
$\rhd$ $(G.c := d).(z \leq Z.e.(G.e))$
$\equiv$ { case analysis; substitution; target assertion $z \leq Z.e.(G.e)$ }
$c = e \Rightarrow z \leq Z.c.d$
$\Leftarrow$ { precondition $J \leq Z.c.d$ }
$c = e \Rightarrow z \leq J$
$\equiv$ { invariant P4: $z \leq J$ }
true
$]|$

$|[\ m \leq Z.e.(G.e) \wedge J \leq Z.c.d$
$\rhd$ $(G.c := d).(m \leq Z.e.(G.e))$
$\equiv$ { case analysis; substitution; target assertion $m \leq Z.e.(G.e)$ }
$c = e \Rightarrow m \leq Z.c.d$
$\Leftarrow$ { precondition $J \leq Z.c.d$ }
$c = e \Rightarrow m \leq J$
$\equiv$ { assume co-assertion $m \leq J$ (see postcondition $m < J$ of assignment $m := J\text{-}1$) }
true
$]|$

$|[\ Z.e.(G.e) \leq Z.C.(G.C) \wedge J \leq Z.c.d$
$\rhd$ $(G.c := d).(Z.e.(G.e) \leq Z.C.(G.C))$
$\equiv$ { case analysis; substitution; target assertion $Z.e.(G.e) \leq Z.C.(G.C)$ }
$(c = e \wedge c = C \Rightarrow Z.c.d \leq Z.c.d)$
$\wedge (c = e \wedge c \neq C \Rightarrow Z.c.d \leq Z.C.(G.C))$
$\wedge (c \neq e \wedge c = C \Rightarrow Z.e.(G.e) \leq Z.c.d)$
$\equiv$ { reflexivity $\leq$ }
$(c = e \wedge c \neq C \Rightarrow Z.c.d \leq Z.C.(G.C))$
$\wedge (c \neq e \wedge c = C \Rightarrow Z.e.(G.e) \leq Z.c.d)$
$\Leftarrow$ { precondition $J \leq Z.c.d$ }
$(c = e \wedge c \neq C \Rightarrow Z.c.d \leq Z.C.(G.C))$
$\wedge (c \neq e \wedge c = C \Rightarrow Z.e.(G.e) \leq J)$
$\equiv$ { invariant P1 with $v, w := e, G.e$; topology of component W: $j\text{-}1 \leq J$ }
$c = e \wedge c \neq C \Rightarrow Z.c.d \leq Z.C.(G.C)$
$\Leftarrow$ { calculus, we could prove $Z.c.d = J$ and $J\text{-}1 = Z.C.(G.C)$ }
$c \neq e \vee c = C$
$]|$

We adopt this condition as invariant Q0.

|[J ≤ Z.c.(G.c)
▷ (C := c).(Z.e.(G.e) ≤ Z.C.(G.C))
≡ { substitution }
Z.e.(G.e) ≤ Z.c.(G.c)
⇐ { precondition *J ≤ Z.c.(G.c)* }
Z.e.(G.e) ≤ J
≡ { invariant P1 with *v, w := e, G.e*; topology of component W: *j-1 ≤ J* }
true
]|

Pre: j = 0 ∧ J = 0 ∧ P0 ∧ P1 ∧ P2 ∧ P3 ∧ P4	
W: **loc**: c, d, j, s, x **priv**: C, G, J, Y, Z * [x.j, j := s.x, j+1 ;<c, d : c ≠ E ∧ d ≠ G.c {c ≠ E}> ; {d ≠ G.c} Y.c.d, Z.c.d := x.J, J ; {J ≤ Z.c.d} G.c := d ; {J ≤ Z.c.(G.c)} C := c ; {J ≤ Z.C.(G.C)} J := J+1]	R: **loc**: e, f, m, y, z **priv**: E * [m := J-1 ; {m < J} e := C ; {m ≤ J}{m ≤ Z.e.(G.e)}{z ≤ Z.e.(G.e)} {Z.e.(G.e) ≤ Z.C.(G.C)} E := e ; {m ≤ J}{m ≤ Z.e.(G.e)}{E = e} {z ≤ Z.e.(G.e)}{Z.e.(G.e) ≤ Z.C.(G.C)} <f := G.e {G.E = f}> ; {m ≤ Z.e.f}{E = e}{z ≤ Z.e.f} {Z.e.f ≤ Z.C.(G.C)} y, z := Y.e.f, Z.e.f ; {y = x.z}{z < j}{m ≤ z} print(y)]
Inv: P0: (∀v, w :: Y.v.w = x.(Z.v.w)) P1: (∀v, w :: Z.v.w < j) P2: J-1 ≤ Z.C.(G.C) P3: z ≤ Z.C.(G.C) P4: z ≤ J Q0: **?** c ≠ e ∨ c = C	
Con: C0: *z* is ascending in time C1: no write operation overlaps some operation on the same element of *Y*	

We are left with invariant Q0: $c \neq e \ \vee \ c = C$. For initial correctness we require precondition $c = C$ (although [Hes98] requires $e = C$). Maintenance can only be endangered by assignment $c, d : c \neq E \ \wedge \ d \neq G.c$ in component W, for which we calculate:

$(c, d : c \neq E \ \wedge \ d \neq G.c).Q0$

$\Leftarrow$ { substitution; orthogonality }

$(\ \forall v:\ v \neq E\ :\ v \neq e\ \vee\ v = C\ )$

$\equiv$ { trading, eliminate inequalities }

$(\ \forall v:\ v = e\ :\ v = E\ \vee\ x = C\ )$

$\equiv$ { one-point rule }

$e = E\ \vee\ e = C$

$\equiv$ { case analysis, towards use of Q0 }

$(\ c = e\ \Rightarrow\ e = E \vee c = C\ )\ \wedge\ (\ c \neq e\ \Rightarrow\ e = E \vee e = C\ )$

$\equiv$ { calculus }

$(\ c \neq e \vee e = E \vee c = C\ )\ \wedge\ (\ c = e \vee e = E \vee e = C\ )$

$\equiv$ { invariant Q0; calculus }

$c = e\ \vee\ e = E\ \vee\ e = C$

$\Leftarrow$ { calculus, it turns out to simplify the derivation (number of assertions) }

$c = e\ \vee\ e = E$

We adopt this condition as invariant Q1.

We then deal with that invariant Q1: $c = e\ \vee\ e = E$. For initial correctness we require (like [Hes98]) precondition $e = E$. Maintenance can be endangered by two assignments for which we calculate:

$(c, d : c \neq E \ \wedge \ d \neq G.c).Q1$

$\Leftarrow$ { substitution; orthogonality }

$(\ \forall v:\ v \neq E\ :\ v = e\ \vee\ e = E\ )$

$\equiv$ { trading, eliminate inequalities }

$(\ \forall v::\ v = E\ \vee\ v = e\ \vee\ e = E\ )$

By far the easiest way to fulfill this condition is to require that variables *e*, *E* and *c* are at most two-valued. That is what we shall do. From the program text we can conclude that these variables must also be at least two-valued. Because variable *C*, the indices of array *G* and the first indices of arrays *Y* and *Z* must have the same type as variables *e*, *E* and *c*, all these variables will be two-valued.

$(e := C).Q1$

$\equiv$ { substitution }

$c = C\ \vee\ C = E$

$\equiv$ { case analysis, towards use of Q1 }

$(\ e = C\ \Rightarrow\ c = e\ \vee\ e = E\ )$

$\wedge\ (\ e \neq C\ \Rightarrow\ c = C\ \vee\ C = E\ )$

$\equiv$ { invariant Q1 }

$e \neq C\ \Rightarrow\ c = C\ \vee\ C = E$

$\equiv$ { shunting }

$C \neq E\ \Rightarrow\ e = C\ \vee\ c = C$

$\equiv$ { invariant Q0: $c \neq e\ \vee\ c = C$ and hence $c = C\ \equiv\ c = C\ \vee\ c = e$ }

$C \neq E\ \Rightarrow\ e = C\ \vee\ c = C\ \vee\ c = e$

$\equiv$ { use the at most two-valued-ness of variables *e*, *C* and *c* }

true

Thus we obtain:

<table>
<tr><td colspan="2">Pre: j = 0 ∧ J = 0 ∧ P0 ∧ P1 ∧ P2 ∧ P3 ∧ P4 ∧ c = C ∧ e = E</td></tr>
<tr><td>W: loc: c, d, j, s, x
priv: C, G, J, Y, Z
* [x.j, j := s.x, j+1
;<c, d : c ≠ E ∧ d ≠ G.c {c ≠ E}>
; {d ≠ G.c}
Y.c.d, Z.c.d := x.J, J
; {J ≤ Z.c.d}
G.c := d
; {J ≤ Z.c.(G.c)}
C := c
; {J ≤ Z.C.(G.C)}
J := J+1
]</td><td>R: loc: e, f, m, y, z
priv: E
* [m := J-1
; {m < J}
e := C
; {m ≤ J}{m ≤ Z.e.(G.e)}{z ≤ Z.e.(G.e)}
{Z.e.(G.e) ≤ Z.C.(G.C)}
E := e
; {m ≤ J}{m ≤ Z.e.(G.e)}{E = e}
{z ≤ Z.e.(G.e)}{Z.e.(G.e) ≤ Z.C.(G.C)}
<f := G.e {G.E = f}>
; {m ≤ Z.e.f}{E = e}{z ≤ Z.e.f}
{Z.e.f ≤ Z.C.(G.C)}
y, z := Y.e.f, Z.e.f
; {y = x.z}{z < j}{m ≤ z}
print(y)
]</td></tr>
<tr><td colspan="2">Inv: P0: (∀v, w :: Y.v.w = x.(Z.v.w)) P4: z ≤ J
P1: (∀v, w :: Z.v.w < j) Q0: c ≠ e ∨ c = C
P2: J-1 ≤ Z.C.(G.C) Q1: c = e ∨ e = E
P3: z ≤ Z.C.(G.C)</td></tr>
<tr><td colspan="2">Con: C0: z is ascending in time
C1: no write operation overlaps some operation on the same element of Y</td></tr>
</table>

We now remove all annotation from our program and remove recursively all variables that are not used. Variable *x* need not to be an array now, so we everywhere replace *x.j* by single variable *x* and replace assignment *x.j, j := s.x, j+1* by statement *produce(x)*.

Observe that the mutually independent conjuncts *c = C* and *e = E* of the precondition can be considered as initial values of local variables *c* and *e* respectively. Because these local variables are assigned a value before they are inspected, we can omit this initial value. Thus we do not need these conjuncts in the precondition.

<table>
<tr><td>W: loc: c, d, x
priv: C, G, Y
* [produce(x)
; c, d : c ≠ E ∧ d ≠ G.c
; Y.c.d := x
; G.c := d
; C := c
]</td><td>R: loc: e, f, y
priv: E
* [e := C
; E := e
; f := G.e
; y := Y.e.f
; print(y)
]</td></tr>
</table>

10.6 Comparison with [Hes98]

If we compare this solution with the solution of [Hes98], we observe that

- he needs a precondition;
- he does not need variable *d*;
- he assumes that variable *f*, the elements of array *G* and the second indices of array *Y* are two-valued.

We will show that we can eliminate variable *d* from our solution under this extra two-valuedness assumption.

Observe that we can add assertion $d \neq G.c$ as precondition of each inspection of variable *d* (Private Variables). If we require that variable *d* is two-valued (and hence variables *f*, the elements of array *G* and the second indices of array *Y* are two-valued), then at these places variable *d* is an alias of $\neg G.c$. Thus we can safely eliminate variable *d* and everywhere replace variable *d* by $\neg G.c$. Because variable *d* is a local variable of component W and thus only occurs in statements of that component, we do not have to reconsider one-pointness of any statements (Private Occurrences).

Thus we obtain, apart from the precondition, the algorithm we already knew from [Hes98]:

```
W: loc:   c, x
   priv: C, G, Y
   * [ produce(x)
     ; c := ¬E
     ; Y.c.(¬G.c) := x
     ; G.c := ¬G.c
     ; C := c
     ]
```

```
R: loc:  e, f, y
   priv: E
   * [ e := C
     ; E := e
     ; f := G.e
     ; y := Y.e.f
     ; print(y)
     ]
```

11. Wait-free write-all problem

This chapter describes a derivation of a non-deterministic wait-free algorithm for the Write-All problem based on the deterministic wait-free algorithm described in [GHMV01].

11.0 Specification

An algorithm for the write-all problem uses a non-empty set of components to write a given value to all positions of a shared array. It is in fact an abstraction of the execution of many independent subtasks by several components.

To develop a formal specification, we consider a non-empty set of components C and a shared array X with set of indices H. As a convention, for dummy variable c we always require $c \in C$.

Without loss of generality, we may assume that array X is a boolean array and that *true* is the value to be written to all its positions. We require as postcondition that all elements in array X have value *true*. Because the wait-free write-all problem considers halting failures of the components, we require that postcondition for each individual component. This means that if at least one component terminates, the postcondition is established.

	con: H **shar**: X
Comp.c:	… {**?** (∀j: j ∈ H: X.j)}

Specification

The problem thus specified has to be solved under two additional constraints, to wit

- that the final algorithm be entirely expressed in one-point atomic statements;
- that the final algorithm be wait-free (see chapter 4).

11.1 Skeleton solution

One may now wonder whether we could use standard sequential solution

for all j: j ∈ H **do** X.j := true **od** {(∀j: j ∈ H: X.j)}

Although it is a correct solution, it does not benefit from the fact that components can co-operate. We will derive a solution that allows co-operation.

As will become clear later, it is convenient to derive a solution that can easily be transformed into a recursive solution. Therefore we generalise constant H into a value parameter $h: h \subseteq H$ of a fresh procedure *traverse*. The components then become *traverse(H)*, if we require specification: {h ⊆ H} traverse(h) {**?** (∀j: j ∈ h: X.j)}

To enable co-operation, we introduce fresh shared variable M with invariants P0: $M \subseteq H$ and P1: $(\forall j: j \in M: X.j)$. Thanks to invariant P1, postcondition $(\forall j: j \in h: X.j)$ of procedure *traverse* is implied if we replace it by postcondition $h \subseteq M$.

	con: H **shar**: M, X
Comp.c:	traverse(H) {(∀j: j ∈ H: X.j)}
traverse(h) =	{h ⊆ H} … {**?** h ⊆ M}
Inv: P0:	**?** M ⊆ H P1: **?** (∀j: j ∈ M: X.j)

In our solutions, global correctness of all assertions except the system invariants turns out to be guaranteed by the Rule of Widening. Therefore we will hardly explicitly deal with global correctness in our derivation.

We first deal with assertion $h \subseteq M$. Because h is a value parameter, that assertion is most easily established by an assignment $M := h$ or $M := H$. However, these are far too demanding for maintenance of invariant P1, so we will extend set M in smaller steps. We apply the sequential programming technique of replacing constant h by fresh local variable g: we introduce a repetition with guard $g \subset h$ and repetition invariants $g \subseteq h$ and $g \subseteq M$. Because the multiprogram ought to be wait-free, we must guarantee termination of that repetition. Therefore we impose variant function $| h \setminus g |$.

We also deal with invariants P0: $M \subseteq H$ and P1: $(\forall j: j \in M: X.j)$. For initial correctness we require precondition $M = \varnothing$. Maintenance will be guaranteed in what follows.

Pre:	M = ∅ **con**: H **shar**: M, X
Comp.c:	traverse(H) {(∀j: j ∈ H: X.j)}
traverse(h) =	|[**var**: g {h ⊆ H} … {Inv: **?** g ⊆ h}{Inv: **?** g ⊆ M}{vf: **?** | h \ g |} **do** g ⊂ h → … **od** {h ⊆ M}]|
Inv: P0:	M ⊆ H P1: (∀j: j ∈ M: X.j)

We first deal with variant function $| h \setminus g |$. Boundedness is guaranteed by definition of the size of a set. Using repetition invariant $g \subseteq h$, decrease is guaranteed by inserting statement $\{? \; g \subset g'\} \;\; g := g'$, for g' a fresh local variable, in the body of the repetition if we ensure that there is no other assignment to variable g in the body of the repetition.

We also deal with both repetition invariants $g \subseteq h$ and $g \subseteq M$. For initial correctness we insert an assignment $g: g \subseteq h \cap M$. For maintenance we require preconditions $g' \subseteq h$ and $g' \subseteq M$ of assignment $g := g'$.

Pre:	M = ∅ **con**: H **shar**: M, X		
Comp.c:	traverse(H) {(∀j: j ∈ H: X.j)}		
traverse(h) =	|[**var**: g, g' {h ⊆ H} g: g ⊆ h ∩ M ; {Inv: g ⊆ h}{Inv: g ⊆ M}{vf: | h \ g |} **do** g ⊂ h → … ; {**?** g ⊂ g'}{**?** g' ⊆ h}{**?** g' ⊆ M} g := g' **od** {h ⊆ M}]|		
Inv:	P0: M ⊆ H	P1:	(∀j: j ∈ M: X.j)
Con:	No more assignment to variable *g* may be inserted in the body of the repetition.		

To keep the program fragments simple, we now focus on the remaining queried assertions:

	{h ⊆ H}{g ⊂ h}{g ⊆ M} … {**?** g ⊂ g'}{**?** g' ⊆ h}{**?** g' ⊆ M}		
Inv:	P0: M ⊆ H	P1:	(∀j: j ∈ M: X.j)
Con:	No assignment to variable *g* may be inserted.		

Because each of the assertions $g \subset g'$, $g' \subseteq h$ and $g' \subseteq M$ contains variable g', we insert an assignment $g': g \subset g' \subseteq h \cap M$. To ensure that it has a solution we require precondition $g \subset h \cap M$ of that assignment.

That precondition $g \subset h \cap M$ of the assignment to g' is not yet implied by guard $g \subset h$ and repetition invariant $g \subseteq M$. Because variable h is a value parameter and we may not insert an assignment to variable g, we have hardly any choice but inserting an assignment to variable M. To do so without endangering other assertions (not the system invariants) about M, we will consider an extension of set M. We introduce fresh variable m and insert assignment $M := M \cup m$ for which we require precondition $g \subset h \cap (M \cup m)$. For maintenance of invariants P0 and P1 we also require preconditions $m \subseteq H$ and $(\forall j: j \in m: X.j)$ respectively of that assignment.

		|[**var**: m {h ⊆ H}{g ⊂ h}{g ⊆ M} … ; {**?** g ⊂ h ∩ (M ∪ m)}{**?** m ⊆ H}{**?** (∀j: j ∈ m: X.j)} M := M ∪ m ; {g ⊂ h ∩ M} g': g ⊂ g' ⊆ h ∩ M {g ⊂ g'}{g' ⊆ h}{g' ⊆ M}]|		
Inv:	P0:	M ⊆ H	P1:	(∀j: j ∈ M: X.j)
Con:	No assignment to variable *g* may be inserted.			

We first establish assertion *(∀j: j ∈ m: X.j)* by inserting statement ***for all*** *j: j ∈ m* ***do*** *X.j := true* ***od*** for which we require precondition $m \subseteq H$. Note that that statement does not violate invariants or assertions in other components (Widening). We establish postconditions $g \subset h \cap (M \cup m)$ and $m \subseteq H$ of that statement by requiring them as preconditions of that statement.

		|[**var**: m {h ⊆ H}{g ⊂ h}{g ⊆ M} … ; {**?** g ⊂ h ∩ (M ∪ m)}{**?** m ⊆ H} **for all** j: j ∈ m **do** X.j := true **od** ; {g ⊂ h ∩ (M ∪ m)}{m ⊆ H}{(∀j: j ∈ m: X.j)} M := M ∪ m ; {g ⊂ h ∩ M} g': g ⊂ g' ⊆ h ∩ M {g ⊂ g'}{g' ⊆ h}{g' ⊆ M}]|		
Inv:	P0:	M ⊆ H	P1:	(∀j: j ∈ M: X.j)
Con:	No assignment to variable *g* may be inserted.			

We are left with assertions $g \subset h \cap (M \cup m)$ and $m \subseteq H$. We want to establish them by inserting an assignment to *m*, but we first simplify (and slightly strengthen) these conditions, to reduce the number of different variables:

|[h ⊆ H ∧ g ⊂ h ∧ g ⊆ M
▷ g ⊂ h ∩ (M ∪ m) ∧ m ⊆ H
⇐ { precondition $h \subseteq H$, eliminate variable H }
g ⊂ h ∩ (M ∪ m) ∧ m ⊆ h
≡ { distribute ∩ over ∪ }
g ⊂ (h ∩ M) ∪ (h ∩ m) ∧ m ⊆ h
≡ { conjunct $m \subseteq h$ and hence $h \cap m = m$ }
g ⊂ (h ∩ M) ∪ m ∧ m ⊆ h
⇐ { guard $g \subset h$ and repetition invariant $g \subseteq M$, and hence $g \subseteq h \cap M$ }
g ⊂ g ∪ m ∧ m ⊆ h
]|

We insert statement $m: g \subset g \cup m \ \wedge \ m \subseteq h$. Guard $g \subset h$ ensures that this assignment has a solution.

We now return to the full program. Thus we obtain:

Pre:	M = ∅ **con**: H **shar**: M, X
Comp.c:	traverse(H) {(∀j: j ∈ H: X.j)}
traverse(h) =	|[**var**: g, g', m {h ⊆ H} g: g ⊆ h ∩ M ; {Inv: g ⊆ h}{Inv: g ⊆ M}{vf: | h \ g |} **do** g ⊂ h → {g ⊂ h}{g ⊆ M} m: g ⊂ g ∪ m ∧ m ⊆ h ; {g ⊂ h ∩ (M ∪ m)}{m ⊆ H} **for all** j: j ∈ m **do** X.j := true **od** ; {g ⊂ h ∩ (M ∪ m)}{m ⊆ H}{(∀j: j ∈ m: X.j)} M := M ∪ m ; {g ⊂ h ∩ M} g': g ⊂ g' ⊆ h ∩ M ; {g ⊂ g'}{g' ⊆ h}{g' ⊆ M} g := g' **od** {h ⊆ M}]|
Inv: P0:	M ⊆ H P1: (∀j: j ∈ M: X.j)
Con:	No more assignment to variable *g* may be inserted in the body of the repetition.

Annotated Skeleton Solution

Note that global correctness of all assertions except the system invariants is guaranteed by the Rule of Widening.

Because we did not exploit the recursion yet, we could eliminate procedure *traverse*. Note that, thanks to P0, we have $H \cap M = M$. Thus we would obtain:

Pre:	M = ∅ **con**: H **shar**: M, X
Comp.c:	|[**var**: g, g', m g: g ⊆ M ; **do** g ⊂ H → m: g ⊂ g ∪ m ∧ m ⊆ H ; **for all** j: j ∈ m **do** X.j := true **od** ; M := M ∪ m ; g': g ⊂ g' ⊆ M ; g := g' **od**]|

11.2 Recursive solution

Observe that Hoare-triple {m ⊆ H} **for all** j ∈ m **do** X.j := true **od** {(∀j: j ∈ m: X.j)} is also the specification (see the beginning of section 11.1) of *traverse(m)*. Therefore we transform our solution into a recursive solution and replace that statement by *traverse(m)*. Because the multiprogram must be wait-free, we must guarantee that the recursion terminates. Therefore we impose variant function | *h* |. Boundedness is guaranteed by definition of the size of a set; for decrease we require precondition *m* ⊂ *h* of *traverse(m)*.

Pre:	M = ∅ **con**: H **shar**: M, X
Comp.c:	traverse(H)
traverse(h) =	|[**var**: g, g', m {h ⊆ H} g: g ⊆ h ∩ M ; **do** g ⊂ h → m: g ⊂ g ∪ m ∧ m ⊆ h ; {**?** m ⊂ h} traverse(m) ; M := M ∪ m ; g': g ⊂ g' ⊆ h ∩ M ; g := g' **od**]|

For assertion *m* ⊂ *h* we strengthen conjunct *m* ⊆ *h* in the requirements of the assignment to *m* into conjunct *m* ⊂ *h*. To ensure that that assignment still has a solution we calculate:

```
|[   g ⊂ h
▷       (∃m :: g ⊂ g ∪ m  ∧  m ⊂ h)
  ≡         { calculus }
        1 ≤ | h \ g |  ∧  1 < | h |
  ≡         { guard g ⊂ h and hence 1 ≤ | h \ g | and | h | ≠ 0 }
        | h | ≠ 1
]|
```

We require this condition as precondition of the assignment to *m*.

We are left with precondition | *h* | ≠ *1* of the assignment to *m*. Because *h* is a value parameter, we have hardly any choice but introducing a selection statement with as guarded command the current solution with guard | *h* | ≠ *1*. Because the multiprogram must be wait-free and hence non-blocking, we must also introduce a guarded command with a guard weaker than | *h* | = *1*. We re-introduce the non-co-operative guarded command *true* → ***for all*** *j: j* ∈ *h* ***do*** *X.j := true* ***od*** which meets the specification of *traverse(h)* and is harmless (Widening) to the original solution (see the annotated skeleton solution).

Thus we obtain:

Pre:	M = ∅ **con**: H **shar**: M, X
Comp.c:	traverse(H)
traverse(h) =	{h ⊆ H} **if** true → **for all** j: j ∈ h **do** X.j := true **od** [] | h | ≠ 1 → |[**var**: g, g', m g: g ⊆ h ∩ M ; **do** g ⊂ h → m: g ⊂ g ∪ m ∧ m ⊂ h ; {m ⊂ h} traverse(m) ; M := M ∪ m ; g': g ⊂ g' ⊆ h ∩ M ; g := g' **od**]| **fi**

Recursive Solution

11.3 Improving the efficiency

Note that precondition $h \subseteq H$ of *traverse(h)*, which is equivalent to $h \in P(H)$, means that each subset of *H* can be used as parameter of *traverse*. Therefore each element of array *X* may be set to *true* very often, thus leading to inefficiency. To restrict the possible parameters, we introduce a binary relation *Z* on subsets of *H*.

Before we continue, we first introduce some notation. For any binary relation *R* on any set *A* we will associate sets R_a for $a \in A$, satisfying $b \in R_a \equiv (a, b) \in R$. We will need the reflexive transitive closure of a relation *R*, to be denoted by relation R^*.

We now strengthen strengthen precondition $h \subseteq H$ of *traverse(h)* using relation *Z*. Because of the recursive calls to *traverse*, we strengthen it into $h \in Z^*_H$. In what follows we will collect properties of relation *Z*.

Re: $h \in Z^*_H$

To establish this precondition of *traverse(h)*, we will exploit that Z^* is the reflexive transitive closure of relation *Z*. For initial call *traverse(H)* condition $H \in Z^*_H$ is guaranteed by reflexivity of Z^*; for recursive call *traverse(m)* we calculate:

|[$h \in Z^*_H$
▷ $m \in Z^*_H$
⇐ { relation Z^* is a transitive closure of relation Z }
$m \in Z_h \wedge h \in Z^*_H$
≡ { use $h \in Z^*_H$ (by induction) }
$m \in Z_h$
]|

We require this condition as precondition of recursive call *traverse(m)*, i.e. – see program text – we require: $m: g \subset g \cup m \wedge m \subset h \ \{? \ m \in Z_h\}$.
Thus we use relation Z to relate next recursive parameter m to current parameter h.

Re: $\{h \in Z^*_H\}\{|\ h\ | \neq 1\}\ m: g \subset g \cup m \wedge m \subset h \ \{? \ m \in Z_h\}$
To establish this postcondition we strengthen conjunct $m \subset h$ of this assignment into $m \in Z_h$, which is a strengthening because we require the following property of Z:

Q0: $(\forall r: r \in Z^*_H \wedge |\ r\ | \neq 1: (\forall s: s \in Z_r : s \subset r))$.

Thus we obtain assignment $m: g \subset g \cup m \wedge m \in Z_h$. To ensure that it has a solution, we require it to have precondition $(\exists m:: g \subset g \cup m \wedge m \in Z_h)$, i.e. we require
$g \subset h \rightarrow \{? \ (\exists m:: g \subset g \cup m \wedge m \in Z_h)\}$.

Re: $\{h \in Z^*_H\}\{|\ h\ | \neq 1\}\ g \subset h \rightarrow \{? \ (\exists m:: g \subset g \cup m \wedge m \in Z_h)\}$
For postcondition $(\exists m:: g \subset g \cup m \wedge m \in Z_h)$ of guard $g \subset h$ we calculate a precondition:
|[$h \in Z^*_H \wedge |\ h\ | \neq 1 \wedge g \subset h$
▷ $(\exists m:: g \subset g \cup m \wedge m \in Z_h)$
≡ { trading }{ set calculus }
$(\exists m: m \in Z_h: \neg(m \subseteq g))$
≡ { De Morgan }
$\neg(\forall m: m \in Z_h: m \subseteq g)$
≡ { set calculus }
$\neg((\cup m: m \in Z_h: m) \subseteq g)$
⇐ { use $g \subset h$, eliminate variable g }
$\neg((\cup m: m \in Z_h: m) \subset h)$
⇐ { set calculus, eliminate the negation. (Using Q0 we could prove that this step is an equivalence.) }
$h = (\cup m: m \in Z_h: m)$
]|

We require this condition as property Q1 of relation Z:

Q1: $(\forall r: r \in Z^*_H \wedge |\ r\ | \neq 1: r = (\cup s: s \in Z_r: s))$.

Pre:	$M = \emptyset \ \wedge$ Q0..1 **con**: H, Z **shar**: M, X
Comp.c:	traverse(H)
traverse(h) =	$\{h \in Z^*_H\}$ **if** true $\rightarrow$ **for all** j: j ∈ h **do** X.j := true **od** [] | h | ≠ 1 → |[**var**: g, g', m g: g ⊆ h ∩ M ; **do** g ⊂ h → {(∃m:: g ⊂ g ∪ m ∧ m ∈ Z_h)} m: g ⊂ g ∪ m ∧ m ∈ Z_h ; {m ∈ Z_h} traverse(m) ; M := M ∪ m ; g': g ⊂ g' ⊆ h ∩ M ; g := g' **od**]| **fi**

To simplify this multiprogram, we want to replace assignment $\{h \in Z^*_H\}\{|\ h\ | \neq 1\}$ *m:* $g \subset g \cup m \ \wedge \ m \in Z_h$ by an assignment *m:* $m \in f$, for fresh local variable *f*. For each of the two conjuncts in the original assignment, we now calculate a precondition for this replacement:

|[$h \in Z^*_H \ \wedge \ |\ h\ | \neq 1 \ \wedge \ m \in f$
▷ $m \in Z_h$
⇐ { use $m \in f$ }
$f \subseteq Z_h$
≡ { assume • $f \subseteq Z_h$ }
true
]|

|[$h \in Z^*_H \ \wedge \ |\ h\ | \neq 1 \ \wedge \ m \in f$
▷ $g \subset g \cup m$
≡ { assume • $g = (\cup t: t \in E: t)$ for some expression E to be determined. Note that we can reverse this step by choosing $E = \{g\}$ }
$(\cup t: t \in E: t) \subset (\cup t: t \in E: t) \cup m$
≡ { set calculus }
$(\cup t: t \in E: t) \subset (\cup t: t \in E \cup \{m\}: t)$
⇐ { assume monotonicity property • Q2 (see below) }
$E \subset E \cup \{m\} \subseteq Z_h$
≡ { set calculus }
$E \subset E \cup \{m\} \ \wedge \ E \subseteq Z_h \ \wedge \ m \in Z_h$
≡ { set calculus }{ use $m \in f$ and hence $m \in Z_h$ (see calculation above) }
$m \notin E \ \wedge \ E \subseteq Z_h$
≡ { choose • $E = Z_h \setminus f$, use $m \in f$ and hence $m \notin E$ }
true
]|

For step “Q2” we require the following monotonicity property of Z:

Q2: $(\forall r: r \in Z^*_H \wedge |r| \neq 1: (\forall s, s': s \subset s' \subseteq Z_r: (\cup t: t \in s: t) \subset (\cup t: t \in s': t)))$.

On account of these calculations we may replace assignment $m: g \subset g \cup m \wedge m \in Z_h$ by $m: m \in f$ if we introduce a fresh local variable f such that the following local invariant (which holds at all control points in the second guarded command of the selection statement of this component) holds:

P2: $g = (\cup t: t \in Z_h \setminus f: t) \wedge f \subseteq Z_h$.

To ensure that this assignment has a solution we require it to have precondition $\varnothing \subset f$.

Invariance of P2: $g = (\cup t: t \in Z_h \setminus f: t) \wedge f \subseteq Z_h$
For initial correctness we replace assignment $g: g \subseteq h \cap M$ by assignment
f, g: $g \subseteq h \cap M \wedge g = (\cup t: t \in Z_h \setminus f: t) \wedge f \subseteq Z_h$,
which has a solution (e.g. $f, g = Z_h, \varnothing$).

For maintenance we introduce fresh local variable f' and replace assignment $g := g'$ by assignment $f, g := f', g'$ for which we require precondition $g' = (\cup t: t \in Z_h \setminus f': t) \wedge f' \subseteq Z_h$, i.e. we require $g': g \subset g' \subseteq h \cap M \; \{? \; g' = (\cup t: t \in Z_h \setminus f': t) \wedge f' \subseteq Z_h\}$.

Re: $g': g \subset g' \subseteq h \cap M \; \{? \; g' = (\cup t: t \in Z_h \setminus f': t) \wedge f' \subseteq Z_h\}$
For this postcondition we replace this assignment by assignment
f', g': $g \subset g' \subseteq h \cap M \wedge g' = (\cup t: t \in Z_h \setminus f': t) \wedge f' \subseteq Z_h$.
Later on we will ensure that it has a solution.

Pre:	$M = \emptyset \ \wedge$ Q0..2 **con**: H, Z **shar**: M, X
Comp.c:	traverse(H)
traverse(h) =	$\{h \in Z^*_H\}$ **if** true $\rightarrow$ **for all** j: j ∈ h **do** X.j := true **od** [] | h | ≠ 1 → |[**var**: f, f', g, g', m f, g: $g \subseteq h \cap M \ \wedge \ g = (\cup t: t \in Z_h \setminus f: t) \ \wedge \ f \subseteq Z_h$; **do** $g \subset h \rightarrow$ {**?** $\emptyset \subset f$} m: m ∈ f ; {$m \in Z_h$} traverse(m) ; $M := M \cup m$; {**?** ∃ solution of the assignment to *f'* and *g'* } f', g': $g \subset g' \subseteq h \cap M$ $\wedge \ g' = (\cup t: t \in Z_h \setminus f': t) \ \wedge \ f' \subseteq Z_h$; $\{g' = (\cup t: t \in Z_h \setminus f': t) \ \wedge \ f' \subseteq Z_h\}$ f, g := f', g' **od**]| **fi**
Inv: P2:	$g = (\cup t: t \in Z_h \setminus f: t) \ \wedge \ f \subseteq Z_h$

It turns out that both queried assertions can be more easily fulfilled, after the elimination of variables *g* and *g'*. Therefore we now want to eliminate these variables, but we must first ensure that the program does not depend on them, i.e. we must ensure that upon elimination of these variables the behaviour of the program does not change. So we have to rewrite the following statements:

- f, g: $g \subseteq h \cap M \ \wedge \ g = (\cup t: t \in Z_h \setminus f: t) \ \wedge \ f \subseteq Z_h$
- $g \subset h \rightarrow$
- f', g': $g \subset g' \subseteq h \cap M \ \wedge \ g' = (\cup t: t \in Z_h \setminus f': t) \ \wedge \ f' \subseteq Z_h$

Re: $\{h \in Z^*_H\}\{| h | \neq 1\} \ g \subset h \rightarrow \ \{? \ \emptyset \subset f\}$

To ensure that this guard does not depend on variables *g* and *g'*, we calculate an equivalent guard that does not contain these variables:

$|[\quad h \in Z^*_H \ \wedge \ | h | \neq 1$

$\triangleright \quad g \subset h$

$\equiv$ { conjunct $g = (\cup t: t \in Z_h \setminus f: t)$ of invariant P2}{ Q1 with $r := h$ }

$(\cup t: t \in Z_h \setminus f: t) \subset (\cup s: s \in Z_h: s)$

$\equiv$ { "⇐": Q2 with $s, s' := Z_h \setminus f, Z_h$ }{ "⇒": use $Z_h \setminus f \subseteq Z_h$ }

$Z_h \setminus f \subset Z_h$

$\equiv$ { "⇐": conjunct $f \subseteq Z_h$ of invariant P2 }{ "⇒": set calculus}

$\emptyset \subset f$

]|

We replace guard $g \subset h$ by the equivalent guard $\varnothing \subset f$, which no longer contains variables g or g'. Note that this guard establishes required postassertion $\varnothing \subset f$.

Re: $\{h \in Z^*_H\}\{|\, h \,| \neq 1\}\ f, g: g \subseteq h \cap M \ \wedge\ g = (\cup t: t \in Z_h \setminus f: t) \ \wedge\ f \subseteq Z_h$

To ensure that this assignment does not depend on variables g and g', we will rewrite conjunct $g \subseteq h \cap M$ such that it does not contain these variables. Before we start our calculation, we must realise that variables f and g are bound within this assignment. To emphasise this, we rename these variables using substitution $f, g := \alpha, \beta$. We now calculate:

$|[\quad h \in Z^*_H \ \wedge\ |\, h \,| \neq 1 \ \wedge\ \beta = (\cup t: t \in Z_h \setminus \alpha: t) \ \wedge\ \alpha \subseteq Z_h$

$\triangleright\quad \beta \subseteq h \cap M$

$\equiv\quad$ { use $\beta = (\cup t: t \in Z_h \setminus \alpha: t)$ }

$(\cup t: t \in Z_h \setminus \alpha: t) \subseteq h \cap M$

$\Leftarrow\quad$ { introduce fresh shared mapping Y with $\bullet\ (\cup t: t \in Z_h \setminus Y.h: t) \subseteq h \cap M$, to improve the symmetry }

$(\cup t: t \in Z_h \setminus \alpha: t) \subseteq (\cup t: t \in Z_h \setminus Y.h: t)$

$\Leftarrow\quad$ { monotonicity with respect to $\subseteq$ }

$Z_h \setminus \alpha \subseteq Z_h \setminus Y.h$

$\Leftarrow\quad$ { set calculus }

$Y.h \subseteq \alpha$

$]|$

On account of this calculation we may replace this assignment by assignment

$f, g: Y.h \subseteq f \subseteq Z_h \ \wedge\ g = (\cup t: t \in Z_h \setminus f: t)$.

To ensure that it has a solution, we assume $\bullet\ Y.h \subseteq Z_h$. In order to meet both dotted requirements, we introduce a fresh shared mapping $Y: P(H) \rightarrow P(P(H))$ such that the following system invariant holds:

P3: $(\forall r: r \in Z^*_H \ \wedge\ |\, r \,| \neq 1: (\cup t: t \in Z_r \setminus Y.r: t) \subseteq r \cap M \ \wedge\ Y.r \subseteq Z_r)$.

Re: $\{h \in Z^*_H\}\{|\, h \,| \neq 1\}\ f', g': g \subset g' \subseteq h \cap M \ \wedge\ g' = (\cup t: t \in Z_h \setminus f': t) \ \wedge\ f' \subseteq Z_h$

To ensure that this assignment does not depend on variables g and g', we will rewrite conjunct $g \subset g' \subseteq h \cap M$ such that it does not contain these variables. To emphasise that variables f' and g' are bound within this assignment, we rename them using substitution $f', g' := \alpha, \beta$. We now calculate:

$|[\quad h \in Z^*_H \ \wedge\ |\, h \,| \neq 1 \ \wedge\ \beta = (\cup t: t \in Z_h \setminus \alpha: t) \ \wedge\ \alpha \subseteq Z_h$

$\triangleright\quad g \subset \beta \subseteq h \cap M$

$\equiv\quad$ { conjunct $g = (\cup t: t \in Z_h \setminus f: t)$ of P2 }{ use $\beta = (\cup t: t \in Z_h \setminus \alpha: t)$ }

$(\cup t: t \in Z_h \setminus f: t) \subset (\cup t: t \in Z_h \setminus \alpha: t) \subseteq h \cap M$

$\Leftarrow\quad$ { conjunct $(\cup t: t \in Z_h \setminus Y.h: t) \subseteq h \cap M$ of P3, to improve the symmetry }

$(\cup t: t \in Z_h \setminus f: t) \subset (\cup t: t \in Z_h \setminus \alpha: t) \subseteq (\cup t: t \in Z_h \setminus Y.h: t)$

$\Leftarrow\quad$ { Q2 with $s, s' := Z_h \setminus f, Z_h \setminus \alpha$ }{ monotonicity with respect to $\subseteq$ }

$Z_h \setminus f \subset Z_h \setminus \alpha \subseteq Z_h \setminus Y.h$

$\Leftarrow\quad$ { set calculus, use conjunct $f \subseteq Z_h$ of invariant P2 }

$Y.h \subseteq \alpha \subset f$

$]|$

On account of this calculation and conjunct $f \subseteq Z_h$ of invariant P2, we may replace this assignment by assignment $f', g': Y.h \subseteq f' \subset f \ \wedge\ g' = (\cup t: t \in Z_h \setminus f': t)$,
for which we require precondition $Y.h \subset f$ to ensure that it has a solution, i.e. we require

$M := M \cup m$ {**?** Y.h ⊂ f} .

Pre:		M = ∅ ∧ Q0..2 **con**: H, Z **shar**: M, X
Comp.c:		traverse(H)
traverse(h) =		{h ∈ Z^*_H} **if** true → **for all** j: j ∈ h **do** X.j := true **od** [] | h | ≠ 1 → |[**var**: f, f', g, g', m f, g: Y.h ⊆ f ⊆ Z_h ∧ g = (∪t: t ∈ Z_h \ f: t) ; **do** ∅ ⊂ f → {∅ ⊂ f} m: m ∈ f ; {m ∈ Z_h} traverse(m) ; M := M ∪ m ; {**?** Y.h ⊂ f } f', g': Y.h ⊆ f' ⊂ f ∧ g' = (∪t: t ∈ Z_h \ f': t) ; f, g := f', g' **od**]| **fi**
Inv:	P2:	g = (∪t: t ∈ Z_h \ f: t) ∧ f ⊆ Z_h
	P3:	**?** (∀r: r ∈ Z^*_H ∧ | r | ≠ 1: (∪t: t ∈ Z_r \ Y.r: t) ⊆ r ∩ M ∧ Y.r ⊆ Z_r)

Invariance of P3: $(\forall r: r \in Z^*_H \wedge |r| \neq 1: (\cup t: t \in Z_r \setminus Y.r: t) \subseteq r \cap M \wedge Y.r \subseteq Z_r)$
For initial correctness we have to require precondition $(\forall r: r \in Z^*_H \wedge |r| \neq 1: Y.r = Z_r)$ due to precondition $M = \emptyset$. We will show by induction that *traverse(h)* maintains invariant P3. In the base case, if there are no internal calls, P3 is maintained (Orthogonality). On the assumption that for all $m: m \in Z_h$ we have that *traverse(m)* maintains invariant P3, we will ensure that *traverse(h)* maintains invariant P3.

Because of the initial values Z_h of Y, conjunct $f \subseteq Z_h$ of invariant P2 and pending obligation $M := M \cup m$ {**?** Y.h ⊂ f}, we have hardly any choice but to consider assignments to $Y.h$. To do so without endangering assertions $Y.h \subset f$ in other components, we will consider shrinkings of set $Y.h$. Because variable M is not used anymore, assignment $M := M \cup m$ will be removed. Therefore we choose to replace this assignment by assignment $M, Y.h := M \cup m, Y.h \setminus \{v\}$. In order to find a suitable choice for v, we calculate a precondition for maintenance of invariant P3:

|[$h \in Z^*_H \wedge |h| \neq 1$
▷ (M, Y.h := M ∪ m, Y.h \ {v}).P3
≡ { substitution using $h \in Z^*_H$ and $|h| \neq 1$; target invariant P3 and Widening }
$(\cup t: t \in Z_h \setminus (Y.h \setminus \{v\}): t) \subseteq h \cap (M \cup m)$
⇐ { set calculus, towards use of invariant P3 }
$(\cup t: t \in (Z_h \setminus Y.h) \cup \{v\}): t) \subseteq h \cap (M \cup m)$
≡ { set calculus }
$(\cup t: t \in Z_h \setminus Y.h: t) \cup v \subseteq h \cap (M \cup m)$
≡ { conjunct $(\cup t: t \in Z_h \setminus Y.h: t) \subseteq h \cap M$ of invariant P3 and Widening }
$v \subseteq h \cap (M \cup m)$
≡ { • choose $v = m$ and hence $v \subseteq M \cup m$ }
$m \subseteq h$
]|

So we choose $v = m$ and require precondition $m \subseteq h$ of assignment *M, Y.h := M ∪ m, Y.h \ {m},* i.e. we require *traverse(m)* {**?** $m \subseteq h$}. Since *m* and *h* are local variables this is fulfilled by precondition $m \in Z_h$ of *traverse(m)* using Q0.

What remains is pending proof obligation $M := M \cup m$ {**?** $Y.h \subset f$} , or rather
M, Y.h := M ∪ m, Y.h \ {m} {**?** $Y.h \subset f$} .

Re: *M, Y.h := M ∪ m, Y.h \ {m}* {**?** $Y.h \subset f$}
For this postcondition we require precondition $Y.h \setminus \{m\} \subset f$ of this assignment, i.e. we require *traverse(m)* {**?** $Y.h \setminus \{m\} \subset f$} .

Re: $\{h \in Z^*_H\}\{|h| \neq 1\}\{m \in Z_h\}$ *traverse(m)* {**?** $Y.h \setminus \{m\} \subset f$}
Because by precondition $m \in Z_h$ and hence (using Q0) $m \subset h$ shared variable *Y.h* does not occur in *traverse(m),* we require postcondition $Y.h \setminus \{m\} \subset f$ as a precondition of *traverse(m),* i.e. we require *m:* $m \in f$ {**?** $Y.h \setminus \{m\} \subset f$} .

Re: *m:* $m \in f$ {**?** $Y.h \setminus \{m\} \subset f$}
For this postcondition we require precondition $Y.h \subseteq f$ of this assignment, i.e. we require
$\varnothing \subset f \rightarrow$ {**?** $Y.h \subseteq f$} .

Re: $\varnothing \subset f \rightarrow$ {**?** $Y.h \subseteq f$}
To establish condition $Y.h \subseteq f$, we require it as a repetition invariant. Initial correctness of that repetition invariant is guaranteed by assignment $f, g: Y.h \subseteq f \subseteq Z_h \wedge g = (\cup t: t \in Z_h \setminus f: t)$; for invariance we require precondition $Y.h \subseteq f'$ of assignment $f, g := f', g'$, i.e. we require $f', g': Y.h \subseteq f' \subset f \wedge g' = (\cup t: t \in Z_h \setminus f': t)$ {**?** $Y.h \subseteq f'$}, which is obviously fulfilled.

Now we can eliminate variables *g, g'* and *M*. Thus we obtain:

Pre:	$(\forall r: r \in Z^*_H \wedge \lvert r \rvert \neq 1: Y.r = Z_r) \wedge Q0..2$ **con**: H, Z **shar**: X, Y
Comp.c:	traverse(H)
traverse(h) =	$\{h \in Z^*_H\}$ **if** true $\rightarrow$ **for all** j: j ∈ h **do** X.j := true **od** [] | h | ≠ 1 → |[**var**: f, f', m f: Y.h ⊆ f ⊆ Z_h ; **do** ∅ ⊂ f → m: m ∈ f ; {m ∈ Z_h} traverse(m) ; Y.h := Y.h \ {m} ; f': Y.h ⊆ f' ⊂ f ; f := f' **od**]| **fi**
Inv: P2:	… ∧ f ⊆ Z_h
P3:	$(\forall r: r \in Z^*_H \wedge \lvert r \rvert \neq 1: \ldots \wedge Y.r \subseteq Z_r)$

We still have to find a relation *Z* satisfying

Q0: $(\forall r: r \in Z^*_H \wedge \lvert r \rvert \neq 1: (\forall s: s \in Z_r : s \subset r))$
Q1: $(\forall r: r \in Z^*_H \wedge \lvert r \rvert \neq 1: r = (\cup s: s \in Z_r: s))$
Q2: $(\forall r: r \in Z^*_H \wedge \lvert r \rvert \neq 1: (\forall s, s': s \subset s' \subseteq Z_r: (\cup t: t \in s: t) \subset (\cup t: t \in s': t)))$

such that the size of Z^*_H is less than the size of $P(H)$. Note that by Q2, every element of Z_r must contain at least one element of *H* that is not contained in any other element of Z_r; and note that this is also sufficient for Q2. If we require (for efficiency reasons) that the elements of Z_r do not overlap, these four requirements are equivalent to that Z_r, for *r* a non-singleton set, is a partitioning of set *r* into non-empty real parts.

11.4 Implementing sets

In this section we want to implement some sets as arrays. In the first place we will simply treat set Z_h as an array containing exactly the elements of that set. Then we choose to implement local variable *f* as a consecutive part of array Z_h using fresh local variables *i* and *j* satisfying local invariant:

P4: $f = Z_h[i..j] \wedge 0 \leq i \wedge j < \lvert Z_h \rvert$

With the aid of invariant P4, we remove variable *f* from guard $\varnothing \subset f$ by replacing it by guard $i \leq j$ and from assignment *m: m ∈ f* by replacing it by assignment $m: m \in Z_h[i..j]$.

Invariance of P4: $f = Z_h[i..j] \ \wedge \ 0 \le i \ \wedge \ j < |\, Z_h \,|$

For initial correctness we replace assignment $f: Y.h \subseteq f \subseteq Z_h$ by assignment

$$f, i, j: Y.h \subseteq f \subseteq Z_h \ \wedge \ f = Z_h[i..j] \ \wedge \ 0 \le i \ \wedge \ j < |\, Z_h \,|\ ,$$

which has a solution (e.g. $f, i, j = Z_h, 0, |Z_h|-1$, using that the original assignment has a solution and hence $Y.h \subseteq Z_h$).

For maintenance we introduce fresh variables i' and j' and replace assignment $f := f'$ by assignment $f, i, j := f', i', j'$ for which we require precondition $f' = Z_h[i'..j'] \ \wedge \ 0 \le i' \ \wedge \ j' < |\, Z_h \,|$, i.e. we require $f': Y.h \subseteq f' \subset f \ \{?\ f' = Z_h[i'..j'] \ \wedge \ 0 \le i' \ \wedge \ j' < |\, Z_h \,|\}$

Re: $\{h \in Z^*_H\}\{|\, h \,| \neq 1\}\ f': Y.h \subseteq f' \subset f \ \{?\ f' = Z_h[i'..j'] \ \wedge \ 0 \le i' \ \wedge \ j' < |\, Z_h \,|\}$

For this postcondition we replace this assignment by assignment

$$f', i', j': Y.h \subseteq f' \subset f \ \wedge \ f' = Z_h[i'..j'] \ \wedge \ 0 \le i' \ \wedge \ j' < |\, Z_h \,|$$

To ensure that it has a solution we introduce fresh shared mappings L and $R: P(H) \to N$ satisfying invariant

$$\text{P5:} \quad (\ \forall r: r \in Z^*_H \ \wedge \ |\, r \,| \neq 1: Y.r = Z_r[L.r \,..\, R.r] \ \wedge \ 0 \le L.r \ \wedge \ R.r < |\, Z_r \,|\ )$$

then this assignment has a solution (e.g. $f', i', j' = Y.h, L.h, R.h$ using that the original assignment has a solution and hence $Y.h \subset f$).

Invariance of P5: $(\forall r: r \in Z^*_H \ \wedge \ |\, r \,| \neq 1: Y.r = Z_r[L.r \,..\, R.r] \ \wedge \ 0 \le L.r \ \wedge \ R.r < |\, Z_r \,|\ )$

For initial correctness we have to require precondition $(\ \forall r: r \in Z^*_H \ \wedge \ |\, r \,| \neq 1: L.r = 0 \ \wedge \ R.r = |Z_r|-1)$ due to precondition $(\ \forall r: r \in Z^*_H \ \wedge \ |\, r \,| \neq 1: Y.r = Z_r\ )$. For maintenance we replace assignment $Y.h := Y.h \setminus \{m\}$ by assignment $L.h, R.h, Y.h := x, y, Y.h \setminus \{m\}$ for x and y yet to be determined, for which we now calculate a precondition:

$$
\begin{array}{ll}
|[& h \in Z^*_H \ \wedge \ |\, h \,| \neq 1 \\
\rhd & \quad (L.h, R.h, Y.h := x, y, Y.h \setminus \{m\}).P5 \\
\equiv & \qquad \{ \text{ substitution using } h \in Z^*_H \text{ and } |\, h \,| \neq 1; \text{ target invariant P5 } \} \\
 & \quad Y.h \setminus \{m\} \ = \ Z_h[x..y] \ \wedge \ 0 \le x \ \wedge \ y < |\, Z_h \,| \\
\equiv & \qquad \{ \text{ conjunct } Y.h = Z_h[L.h \,..\, R.h] \text{ of invariant P5 } \} \\
 & \quad (\ Z_h[L.h \,..\, R.h]\ ) \setminus \{m\} \ = \ Z_h[x..y] \ \wedge \ 0 \le x \ \wedge \ y < |\, Z_h \,| \\
\Leftarrow & \qquad \{ \text{ conjuncts } 0 \le L.h \text{ and } R.h < |\, Z_h \,| \text{ of invariant P5 } \} \\
 & \quad (\ Z_h[L.h \,..\, R.h]\ ) \setminus \{m\} \ = \ Z_h[x..y] \ \wedge \ L.h \le x \ \wedge \ y \le R.h \\
\Leftarrow & \qquad \{ \text{ case analysis, introduce fresh local variable } k \text{ } \} \\
 & \quad m = Z_h.k \ \wedge (\ \ (\ k < L.h \ \wedge \ x = L.h \ \ \wedge \ y = R.h \ \ ) \\
 & \qquad\qquad \vee (\ k = L.h \ \wedge \ x = L.h + 1 \ \wedge \ y = R.h \ \ ) \\
 & \qquad\qquad \vee (\ R.h = k \ \wedge \ x = L.h \ \ \wedge \ y = R.h - 1\ ) \\
 & \qquad\qquad \vee (\ R.h < k \ \wedge \ x = L.h \ \ \wedge \ y = R.h \ \ )\ ) \\
]| &
\end{array}
$$

So we introduce a fresh local variable k and replace that assignment by the following statement (see also the note below):

{**?** m = Z_h.k}
if k < L.h → {m = Z_h.k}{k < L.h} Y.h := Y.h \ {m}
[] < k = L.h → {m = Z_h.k}{k = L.h} L.h, Y.h := L.h +1, Y.h \ {m} >
[] < R.h = k → {m = Z_h.k}{R.h = k} R.h, Y.h := R.h -1, Y.h \ {m} >
[] R.h < k → {m = Z_h.k}{R.h < k} Y.h := Y.h \ {m}
fi

Note that because global correctness of assertions *k = L.h* and *R.h = k* was not guaranteed, we placed the two corresponding guarded commands within atomic brackets. To ensure that this selection is not a blocking statement, we require precondition $k \leq L.h \ \vee \ R.h \leq k$. So we require *traverse(m)* {**?** m = Z_h.k}{**?** k ≤ L.h ∨ R.h ≤ k} .

Re: {h ∈ Z^*_H}{| h | ≠ 1}{m ∈ Z_h} *traverse(m)* {**?** m = Z_h.k}{**?** k ≤ L.h ∨ R.h ≤ k}
Because by precondition $m \in Z_h$ and hence (using Q0) $m \subset h$ shared variables *L.h* and *R.h* do not occur in *traverse(m)*, we require both postconditions as preconditions of *traverse(m)*, i.e. we require *m:* $m \in Z_h[i..j]$ {**?** m = Z_h.k}{**?** k ≤ L.h ∨ R.h ≤ k}.

Re: {h ∈ Z^*_H}{| h | ≠ 1} *m:* $m \in Z_h[i..j]$ {**?** m = Z_h.k}{**?** k ≤ L.h ∨ R.h ≤ k}
To establish these postconditions we replace this assignment by assignment
k, m: i ≤ k ≤ j ∧ m = Z_h.k ∧ (k ≤ L.h ∨ R.h ≤ k)
To ensure that it has a solution we require preconditions $i \leq j$ and $i \leq L.h \ \vee \ R.h \leq j$ of this assignment, i.e. we require: i ≤ j → {**?** i ≤ j}{**?** i ≤ L.h ∨ R.h ≤ j} .

Re: i ≤ j → {**?** i ≤ j}{**?** i ≤ L.h ∨ R.h ≤ j}
Postcondition $i \leq j$ follows directly from this guard. To establish postcondition $i \leq L.h \ \vee \ R.h \leq j$ we require it as repetition invariant.

Pre:		$(\forall r: r \in Z^*_H \wedge \mid r \mid \neq 1: Y.r = Z_r \wedge L.r = \varnothing \wedge R.r = \mid Z_r \mid -1) \wedge Q0..2$ **con**: H, Z **shar**: L, R, X, Y
Comp.c:		traverse(H)
traverse(h) =		$\{h \in Z^*_H\}$ **if** true $\rightarrow$ **for all** j: $j \in h$ **do** X.j := true **od** [] $\mid h \mid \neq 1 \rightarrow$ $\mid[$ **var**: f, f', i, i', j, j', m f, i, j: $Y.h \subseteq f \subseteq Z_h \wedge f = Z_h[i..j] \wedge 0 \leq i \wedge j < \mid Z_h \mid$; {Inv: **?** $i \leq L.h \vee R.h \leq j$} **do** $i \leq j \rightarrow$ $\{i \leq j\}\{i \leq L.h \vee R.h \leq j\}$ k, m: $i \leq k \leq j \wedge m = Z_h.k \wedge (k \leq L.h \vee R.h \leq k)$; $\{m \in Z_h\}\{m = Z_h.k\}\{k \leq L.h \vee R.h \leq k\}$ traverse(m) ; $\{m = Z_h.k\}\{k \leq L.h \vee R.h \leq k\}$ **if** $k < L.h \rightarrow Y.h := Y.h \setminus \{m\}$ [] $< k = L.h \rightarrow L.h, Y.h := L.h + 1, Y.h \setminus \{m\} >$ [] $< R.h = k \rightarrow R.h, Y.h := R.h - 1, Y.h \setminus \{m\} >$ [] $R.h < k \rightarrow Y.h := Y.h \setminus \{m\}$ **fi** ; f', i', j': $Y.h \subseteq f' \subset f \wedge f' = Z_h[i'..j'] \wedge 0 \leq i' \wedge j' < \mid Z_h \mid$; $\{f' = Z_h[i'..j'] \wedge 0 \leq i' \wedge j' < \mid Z_h \mid\}$ f, i, j := f', i', j' **od** $]\mid$ **fi**
Inv:	P4:	$f = Z_h[i..j] \wedge 0 \leq i \wedge j < \mid Z_h \mid$
	P5:	$(\forall r: r \in Z^*_H \wedge \mid r \mid \neq 1: Y.r = Z_r[L.r .. R.r] \wedge 0 \leq L.r \wedge R.r < \mid Z_r \mid)$

It turns out that the queried repetition invariant can be more easily fulfilled, after the elimination of variables *f, f', m* and *Y*. Therefore we now want to eliminate these variables, but we must first ensure that the program does not depend on them. So we have to rewrite the following statements:

- f, i, j: $Y.h \subseteq f \subseteq Z_h \wedge f = Z_h[i..j] \wedge 0 \leq i \wedge j < \mid Z_h \mid$
- traverse(m)
- f', i', j': $Y.h \subseteq f' \subset f \wedge f' = Z_h[i'..j'] \wedge 0 \leq i' \wedge j' < \mid Z_h \mid$

Re: $\{h \in Z^*_H\}\{\mid h \mid \neq 1\}$ f, i, j: $Y.h \subseteq f \subseteq Z_h \wedge f = Z_h[i..j] \wedge 0 \leq i \wedge j < \mid Z_h \mid$

To ensure that this assignment does not depend on variables *f, f', m* and *Y,* we will rewrite conjunct $Y.h \subseteq f \subseteq Z_h$ such that it does not contain these variables. To emphasise that variables *f'* and *g'* are bound within this assignment, we rename these variables using substitution $f, i, j := \alpha, \beta, \gamma$. We now calculate:

|[$h \in Z^*_H \wedge |h| \neq 1 \wedge \alpha = Z_h[\beta..\gamma] \wedge 0 \leq \beta \wedge \gamma < |Z_h|$
▷ $Y.h \subseteq \alpha \subseteq Z_h$
≡ { use P5: $Y.h = Z_h[L.h .. R.h]$ }{ use conjunct $\alpha = Z_h[\beta..\gamma]$ }
$Z_h[L.h .. R.h] \subseteq Z_h[\beta..\gamma] \subseteq Z_h[0..|Z_h|)$
≡ { calculus, Z_h contains no duplicates }
$0 \leq \beta \leq L.h \wedge R.h \leq \gamma < |Z_h|$
]|

We replace this assignment by assignment $f, i, j: 0 \leq i \leq L.h \wedge R.h \leq j < |Z_h| \wedge f = Z_h[i..j]$.

Re: {m = Z_h.k} traverse(m)

To ensure that this assignment does not depend on variables *f*, *f'*, *m* and *Y*, we replace it by statement *traverse(Z_h.k)* using precondition $m = Z_h.k$.

Re: {$h \in Z^*_H$}{$|h| \neq 1$} f', i', j': $Y.h \subseteq f' \subset f \wedge f' = Z_h[i'..j'] \wedge 0 \leq i' \wedge j' < |Z_h|$

To ensure that this assignment does not depend on variables *f*, *f'*, *m* and *Y*, we will rewrite conjunct $Y.h \subseteq f' \subset f$ such that it does not contain these variables. To emphasise that variables *f'* and *g'* are bound within this assignment, we rename thes variables using substitution $f', i', j' := \alpha, \beta, \gamma$. We now calculate:

|[$h \in Z^*_H \wedge |h| \neq 1 \wedge \alpha = Z_h[\beta..\gamma] \wedge 0 \leq \beta \wedge \gamma < |Z_h|$
▷ $Y.h \subseteq \alpha \subset f$
≡ { use P5: $Y.h = Z_h[L.h .. R.h]$ }{ use $\alpha = Z_h[\beta..\gamma]$ }{ use P4: $f = Z_h[i..j]$ }
$Z_h[L.h .. R.h] \subseteq Z_h[\beta..\gamma] \subset Z_h[i..j]$
≡ { calculus, Z_h contains no duplicates }
$i \leq \beta \leq L.h \wedge R.h \leq \gamma \leq j \wedge (i < \beta \vee \gamma < j)$
]|

On account of this calculation and conjuncts $0 \leq i$ and $j < |Z_h|$ of invariant P4, we may replace this assignment by assignment

$f', i', j': i \leq i' \leq L.h \wedge R.h \leq j' \leq j \wedge (i < i' \vee j' < j) \wedge f' = Z_h[i'..j']$.

Re: $i \leq L.h \vee R.h \leq j$

Using the assignments rewritten above, we can easily deal with repetition invariant $i \leq L.h \vee R.h \leq j$. Initial correctness follows directly from assignment

$f, i, j: 0 \leq i \leq L.h \wedge R.h \leq j < |Z_h| \wedge f = Z_h[i..j]$.

For maintenance we require precondition $i' \leq L.h \vee R.h \leq j'$ of assignment $f, i, j := f', i', j'$, i.e. we require it as postcondition of assignment

$f', i', j': i \leq i' \leq L.h \wedge R.h \leq j' \leq j \wedge (i < i' \vee j' < j) \wedge f' = Z_h[i'..j']$,

which is obviously fulfilled. Note that we have even the stronger repetition invariant $i \leq L.h \wedge R.h \leq j$ (which we will exploit in the next section).

Now we can safely eliminate variables *f*, *f'*, *m* and *Y* and remove some annotation.

Pre:	(∀r: r ∈ Z^*_H ∧ | r | ≠ 1: L.r = 0 ∧ R.r = $\|Z_r\|$-*1*) ∧ Q0..2 **con**: H, Z **shar**: L, R, X
Comp.c:	traverse(H)
traverse(h) =	{h ∈ Z^*_H} **if** true → **for all** j: j ∈ h **do** X.j := true **od** [] | h | ≠ 1 → |[**var**: i, i', j, j', k i, j: 0 ≤ i ≤ L.h ∧ R.h ≤ j < | Z_h | ; {Inv: i ≤ L.h ∨ R.h ≤ j} **do** i ≤ j → k: i ≤ k ≤ j ∧ (k ≤ L.h ∨ R.h ≤ k) ; traverse(Z_h.k) ; **if** k < L.h → **skip** [] < k = L.h → L.h := L.h +1 > [] < R.h = k → R.h := R.h –1 > [] R.h < k → **skip** **fi** ; i', j': i ≤ i' ≤ L.h ∧ R.h ≤ j' ≤ j ∧ (i < i' ∨ j' < j) ; i, j := i', j' **od**]| **fi**

11.5 Grain-size

This solution contains four non-one-point assignments that must finally be eliminated:

- i, j: 0 ≤ i ≤ L.h ∧ R.h ≤ j < \| Z_h \|
- k: i ≤ k ≤ j ∧ (k ≤ L.h ∨ R.h ≤ k)
- **if** … [] … [] … [] … **fi**
- i', j': i ≤ i' ≤ L.h ∧ R.h ≤ j' ≤ j ∧ (i < i' ∨ j' < j)

Re: ***if*** *…* ***[]*** *…* ***[]*** *…* ***[]*** *…* ***fi***

We will start with the selection statement containing four guarded statements. Because it resembles a double compare&swap statement, we will consider the following two smaller selection statements:

{**?** k ≤ L.h}	{**?** R.h ≤ k}
if k < L.h → **skip**	**if** R.h < k → **skip**
[] < k = L.h → L.h := L.h +1 >	[] < R.h = k → R.h := R.h –1 >
fi	**fi**

If we could establish these preassertions, these statements can be implemented as *compare&swap(L.h, k, k+1)* and *compare&swap(R.h, k, k-1)* respectively. We exploit the symmetry by focussing on the version with required precondition *k ≤L.h*, i.e. we require

traverse(Z_h.k) {**?** k ≤ L.h}

Re: $\{h \in Z^*_H\}\{|\, h \,| \neq 1\}$ *traverse(Z_h.k)* {**?** k ≤ L.h}
Because $Z_h.k \in Z_h$ and hence (using Q0) $Z_h.k \subset h$ shared variable *L.h* does not occur in *traverse(Z_h.k)*, we require this postcondition as precondition of *traverse(m)*, i.e. we require
k: $i \leq k \leq j \wedge (k \leq L.h \vee R.h \leq k)$ {**?** k ≤ L.h}.

Re: *k:* $i \leq k \leq j \wedge (k \leq L.h \vee R.h \leq k)$ {**?** k ≤ L.h}
To establish this postcondition, we strengthen this assignment into assignment
k: $i \leq k \leq j \wedge k \leq L.h$
for which we require precondition $i \leq L.h$ to ensure that it has a solution (using that the original assignment has a solution and hence $i \leq j$). As we mentioned in the last section, postcondition $i \leq L.h$ is in fact a repetition invariant, so this condition is established.

So we can replace the complicated selection statement by one of the two compare&swap statements mentioned above. In order to exploit both alternatives, we extend the repetition with an extra guarded command such that each guarded command corresponds to one of the two compare&swap statements.

Re: *i, j:* $0 \leq i \leq L.h \wedge R.h \leq j < |\, Z_h \,|$ and *i', j':* $i \leq i' \leq L.h \wedge R.h \leq j' \leq j \wedge (i < i' \vee j' < j)$
We first deal with the coarse-grained assignment to *i* and *j*. For efficiency reasons we want large solutions for *i* and small solutions for *j*. Because this assignment has a solution and *L.h* and *R.h* are ascending and descending respectively, we may replace it by the parallel composition of assignments *i := L.h* and *j := R.h*, to be denoted as
i := L.h || j := R.h .
Similarly, we can use this construction to replace the sequential composition of the non-one-point assignment to *i'* and *j'* and assignment *i, j := i', j'*.

Re: *k:* $i \leq k \leq j \wedge k \leq L.h$
Although this replaced assignment to *k* is a one-point assignment, we want to clean-up our program a little bit by eliminating variable *k*. Because assignment *k:* $i \leq k \leq j \wedge k \leq L.h$ has a solution, we may replace it by assignment *k := i*. Then we can eliminate variable *k* by everywhere in that guarded command replacing variable *k* by variable *i* (Private Variables).

Thus we obtain:

```
Pre:          ( ∀s: s ⊆ H ∧ | s | ≠ 1:  L.s = 0 ∧ R.s = | Z_s | -1 ) ∧ Q0..2
              con:  H, Z
              shar: L, R, X
Comp.c:       traverse( H )
traverse( h ) = if true      →   for all j: j ∈ h do X.j := true od
                [] | h | ≠ 1 →
                        |[ var: i, j
                          i := L.h  ||  j := R.h
                        ; do i ≤ j →
                                  traverse( Z_h.i )
                                ; compare&swap( L.h, i, i+1 )
                                ; i := L.h  ||  j := R.h
                          [] i ≤ j →
                                  traverse( Z_h.j )
                                ; compare&swap( R.h, j, j-1 )
                                ; i := L.h  ||  j := R.h
                          od
                        ]|
                fi
```

Main Solution

For “non-overlapping” (see section 11.3) relation *Z*, the worst-case time complexity per component is of the order of the size of array *X* (which is optimal), but the worst-case time complexity of the multiprogram is of the order of the size of array *X* times the number of components. The implementation of this solution that is described in [GHMV01] improves on this by reducing the non-determinism in our solution.

An afterthought

Although the derivations of the Skeleton and Recursive solution were rather straightforward, in the two subsequent sections things became more and more complicated. One reason may be that the formulae become more complex. But another reason may be that a main driving force of [GHMV01] was to obtain a good performance, while we were mainly guided by correctness and wait-freeness.

So it is very questionable whether this presentation of such algorithm should be preferred to the presentation of, for example, [GHMV01]. In either case, this derivation contains a lot of examples on how the wait-freeness constraint influences a derivation.

12. Conclusions

The main aim of the project was to investigate whether the method [FvG99] (usually used to derive lock-based multiprograms) could effectively be used to derive non-blocking multiprograms. Because the non-blocking constraints have only a limited (but important) influence on the design, that method could effectively be exploited. However, because performance is not a driving force in that method, the algorithm for the Wait-free write-all problem (chapter 11), which is mainly designed for performance, is more difficult to derive nicely.

Although our presentations of multiprograms are larger in size than the corresponding presentations in the literature, in general they provide more insight in the construction of multiprograms, and: on the fly they provide correctness proofs.

Individual progress of wait-free multiprograms can elegantly be guaranteed using variant functions only, which is in line with the expectations of [FvG99].

Although there are many universal constructions, we only focused on the compare&swap primitive because it entered our derivations more or less automatically. So, whereas [Her93] preferred another universal construction, we have easily derived solutions using the compare&swap primitive.

References

[AJR97] James H. Anderson, Rohit Jain and Srikanth Ramamurthy
Wait-free Object-Sharing Schemes for Real-Time Uniprocessors and Multiprocessors
Proceedings of the IEEE Real-Time Systems Symposium, December 1997

[FvG99] W.H.J. Feijen, A.J.M van Gasteren
On a method of multiprogramming
ISBN 0-387-98870-X

[GHMV01] Jan Friso Groote, Wim H. Hesselink, Sjouke Mauw, Rogier Vermeulen
An algorithm for the asynchronous *Write-All* problem based on process collision
Distributed Computing, Volume 14 (2001), Pages 75-81

[Her90] Maurice Herlihy
A Methodology for Implementing Highly Concurrent Data Structures
Proceedings of the 2nd ACM SIGPLAN,
Volume 25, Issue 3 (March 1990), Pages 197-206

[Her91] Maurice Herlihy
Wait-Free Synchronization
ACM Transactions on Programming Languages and Systems,
Volume 13, Issue 1 (1991), Pages 124-149

[Her93] Maurice Herlihy
A Methodology for Implementing Highly Concurrent Data Objects
ACM Transactions on Programming Languages and Systems,
Volume 15, Issue 5 (1993), Pages 745-770

[Hes98] Wim H. Hesselink
Invariants for the construction of a handshake register
Information Processing Letters, Volume 68 (1998), Pages 173-177

[Int99] Intel Corporation
Intel Architecture Software Developer's Manual 1999

[ST00] Håkan Sundell, Philippas Tsigas
Space Efficient Wait-Free Buffer Sharing in Multiprocessor Real-Time Systems Based on Timing Information
Proceedings of the International Conference on Real-Time Computing Systems and Applications, December 2000