A Methodology for Programming with Concurrency

by

Christian Lengauer

Technical Report CSRG-142

April 1982

Abstract

In this methodology, programming problems which can be specified by an input/output assertion pair are solved in two steps:

(1) Refinement of a correct program that can be implemented sequentially.

(2) Declaration of program properties, "semantic relations", that allow relaxations in the sequencing of the refinement's operations (e.g., concurrency).

Formal properties of refinements comprise semantics (input/output characteristics) and (sequential) execution time. Declarations of semantic relations preserve the semantics but may improve the execution time of a refinement. The consequences are:

(a) The concurrency in a program is deduced from its formal semantics. Semantic correctness is not based on concurrency but precedes it.

(b) Concurrency is a property not of programs but of executions. Programs do not contain concurrent commands, only suggestions (declarations) of concurrency.

(c) The declaration of too much concurrency is impossible. Programs do not contain primitives for synchronization or mutual exclusion.

Acknowledgements

It is not unusual to begin a list of acknowledgements for a Ph.D. with thanks to the supervisor. However, in my case, the person to thank, Eric Hehner, deserves a very special mention:

Rick provided the idea underlying the methodology presented here (the use of declarations to express correct concurrency) while I was still thinking in traditional ways, and in the initial phase of my research he helped me gain confidence in the approach. Without this initial push I would likely not have undertaken this work. Rick's interest and support never wavered. He even invited me to accompany him on his sabbatical leave to Europe - another crucial factor to my success. I feel privileged to have had Rick as supervisor.

The other members of my advisory committee, Allan Borodin, Scott Graham, and Richard Holt also influenced considerably the direction of my research. Last but not least, my external examiner, C.A.R. Hoare, put the final touches on the thesis.

My sincere thanks to all of them; I am leaving this Department of Computer Science with the belief that I have received the best of graduate educations.

Discussions with several other people have helped increase my insight into the properties of my methodology. I would like to stress the contributions of Eike Best, David Gries, Hugh Redelmeier, and Michel Sintzoff.

I gratefully acknowledge financial support by the Social Sciences and Humanities Research Council of Canada and the World University Service of Canada.

One cannot live on work alone. I was fortunate to find in Toronto many nice people to relax and have fun with. In particular, I would like to thank Don, Ben, Ignacio, Ivor, Sam, Mary Anne & Sami, Pierre & Danielle, and Larry & Sherry for their continuing company and friendship which is as much a reward for my stay in Toronto as my degree.

Finally, there are all those who stayed behind: my family back home. Until I am a parent of grown children I shall not comprehend what it takes to see them leave and seek their fortunes on another continent. Wherever I was, my parents and my brother have always been close to me, and I know their love and support will never fail. To them I dedicate this thesis.

To my parents and my brother

Table of Contents

1 Introduction

1.1 Concepts of Concurrent Programs

The last decade has brought considerable advances in the field of programming methodology, in general, and in the understanding of concurrency, in particular.

The popular technique for programming concurrency is to define a set of concurrent units, ***processes***, and to control their interaction by some means of ***synchronization***. The early language constructs proposed in the sixties, **fork** and **join** for processes [Con63] and semaphores for synchronization [Dij68], were intuitive but difficult to formalize. However, the invention of formal methods for the specification of program semantics [Hoa69] increased our understanding and ability to handle process programs.

We commence the program development by specifying, in a ***concurrent command***, a set of processes, $S1, S2, \ldots, Sn$,

$$\textbf{cobegin}\ S1\ //\ S2\ // \ldots //\ Sn\ \ \textbf{coend}$$

which are sequential within themselves but are executed in concurrence with respect to each other. The double slashes (//) signify concurrent execution. To prevent concurrency where it may lead to incorrect program behaviours, we add some construct for ***conditional delay***. For instance,

$$\textbf{await} < B \rightarrow SL >$$

appearing in process Si suspends the execution of Si until logical condition B is satisfied. Typically, B will be generated by some concurrent process Sj, $j \neq i$. Upon validation of B, the execution of Si proceeds with statement list SL. While SL is being executed, any concurrent operation that might cause correctness problems (for example, invalidate B again) is suspended. The technique of forcing activities of different concurrent processes into a sequence in order to preserve correctness is called ***mutual exclusion***. To protect statement S by mutual exclusion, it is framed with angle brackets:

$$< S >$$

These constructs for programming with processes (the concurrent command, conditional delay, and mutual exclusion) can be formally defined [OwGr76a, Lam77], and we can convince ourselves of the correctness of a process program by

(a) first verifying all processes separately as if they were isolated sequential programs, and then

(b) proving the correctness of their interactions in concurrent execution.

A good example is the proof of a concurrent garbage collector [Gri77].

There are, of course, problems and the one we are particularly concerned with in this thesis is the apparent lack of guidelines or criteria to aid the program design. A proof system alone does not necessarily provide support for the development of correct programs. We might continually produce code and discover that it is incorrect. What we need is a ***methodology***, a programming calculus that merges program design and verification in order to obtain correct and, maybe, even particularly suitable programs.

Let us explain why the approach to concurrency just described does not serve as a methodology for program development:

There are no guidelines for the choice of processes. But even if there were, dividing a program into processes is a bold first step, because it usually defines too much concurrency, which then has to be properly pruned with conditional delays and mutual exclusion. Failure to undo all incorrect concurrency leaves one with an incorrect program. Unfortunately, parallel correctness can only be established **after** the development of the processes involved has been completed. Thus the development of processes and the proof of their correct cooperation are strictly separated. One has to understand all process interactions in their entirety in order to arrive at a correct program.

Our goal is a programming methodology that includes aspects of concurrency. In addition to a formally defined language in which one can communicate programs to the computer, we aim at methods for a correct and suitable stepwise development of such programs. In this methodology, concurrency will be a property not of a program but of an execution. If the semantic properties of the program permit concurrency, an implementation should be able to make use of it to whatever extent is possible and practical. But the semantics will determine the concurrency, not vice versa. Then the correctness of the program does not depend on our understanding of concurrency. A consequence of this view is that our programming language will not contain primitives for sequencing, concurrency, or synchronization. These are aspects of executions, not of the program itself.

We will develop and prove concurrency in steps. Each step will increase the concurrency of the program's executions by observing local properties of some program components. A global understanding of the concurrency permitted by the program is not necessary.

We will be result-oriented, i.e., only interested in results of programs and the speed with which these results can be obtained, but not in certain program behaviours. Our programs will suggest suitable computations rather than expressing a set of given computations. In contrast, process programs are behaviour-oriented, i.e., designed with specific computations in mind.

A much-dreaded sign of the complications parallelism introduces into programs is that the complexity of proofs explodes with increasing concurrency. This is blamed on the necessity to argue consistency of shared data. To prevent such an argument, one can either discipline the use of shared variables [Hoa74, OwGr76b], but that restricts also the potential of concurrency. Or one can eliminate shared variables altogether [GCW79, Hoa78b], but may in proofs still have to deal with shared auxiliary variables [AFR80, LeGr81]. ***Auxiliary variables*** are variables added to a program in order to obtain a proof [OwGr76a, OwGr76b]. Our methodology uses shared variables, and subtle concurrency may require a complex proof. But proofs do not contain auxiliary variables.

There are also the obstacles of deadlock and starvation. Deadlock, the situation where the concurrent execution cannot proceed, can occur in terminating as well as non-terminating applications. Criteria for the prevention or avoidance of deadlock have been investigated [Holt72, Lam77, OwGr76a, OwGr76b]. For terminating programs, total correctness implies absence of deadlock. Starvation, the situation where some action concurrent with others can in theory be activated but actually never is, can only occur in non-terminating programs. To avoid starvation one often appeals to a fair scheduler. We shall show that, under certain very simple restrictions, the generalization from terminating to non-terminating programs does not have to pose additional concurrency problems. All programs derived with our methodology will be implicitly free from deadlock and starvation without recourse to an outside authority like a scheduler.

1.2 Devising a Methodology

Our foremost interest is in the result of a program's execution, and in the constraints under which this result can be achieved. In this thesis, we do not consider additional constraints on the program's behaviour.

Consequently, we want to solve programming problems that can be formally specified by an input/output assertion pair. Because we are looking for results, we do not permit programs to generate the false output assertion. This restricts the range of specifiability to finite problems, i.e., problems that have terminating solutions (the false output assertion indicates non-termination). An infinite problem must be expressed by a specifiable finite segment whose terminating solution can be applied repeatedly.

Solutions cannot contain event-driven activities but might be part of a system with real-time constraints. In such a case we may, in addition, specify execution time requirements.

The program development consists of two phases:

(1) Formal refinement of a totally correct program that can be implemented sequentially.

We will use methods that can, if used correctly, only produce refinements whose semantics satisfy the problem specification. The proof of a refinement will also yield a first estimate of its execution time, namely a measure for its sequential execution.

(2) Declaration of program properties, so-called semantic relations, that allow relaxations in the sequencing of the refinement's operations (e.g., concurrency).

We will define semantic relations between refinement components and provide rules for their declaration. Semantic declarations will preserve the semantics of the refinement but may suggest faster executions, e.g., by permitting concurrency.

Semantic declarations are a mechanism for the stepwise development of concurrency. They only require a local understanding of the refinement components appearing in the declared relation. Remember that we are interested in outputs, not in program behaviours. Accordingly, we view concurrency as a tool for satisfying execution time requirements, not for obtaining certain program behaviours.

1.3 Thesis Contribution

There are two different approaches to programming with concurrency:

(a) consider a concurrent world in which sequentiality is the special case,

(b) consider a sequential world in which concurrency is the special case.

We take the latter view: we provide a sequential setting (refinements) and add concurrency (semantic declarations). The refinement calculus is a conglomerate of ideas and concepts previously published, tailored to our needs. The main contribution of this thesis is the way in which concurrency in refinements is expressed and treated.

1.4 Notation

We use the logical operations $\wedge$ (and), $\vee$ (or), $\sim$ (not), $\supset$ (implication), $\equiv$ (equivalence), and quantifiers $\bigwedge_i$ (for all i) and $\bigvee_i$ (there exists i). **N** denotes the natural numbers, **R** the real numbers.

Program properties are described in the weakest precondition calculus. The weakest precondition for statement S with respect to postcondition R (introduced as wp(S,R) in [Dij75, Dij76]) is here denoted $S\{R\}$. We sometimes index statements, e.g., $S_{\mathbf{index}}$ or assertions, e.g., $R_{\mathbf{index}}$. The subscripted weakest precondition is written $S\{R\}_{\mathbf{index}}$. Carefully distinguish $S\{R\}_{\mathbf{index}}$, $S\{R_{\mathbf{index}}\}$, and $S_{\mathbf{index}}\{R\}$!

R^x_E is predicate R with every free occurrence of variable x replaced by expression E. $R^{x_1,\ldots,x_n}_{E_1,\ldots,E_n}$ is R with the free occurrences of all x_k simultaneously replaced by the corresponding E_k.

We will treat an array of variables as a (partial) function. Given a function f, $(f;i:v)$ is a function as f, except that it maps i on v. $(f;i1,\ldots,in:v1,\ldots,vn)$ is as f, except that the images of all i_k are simultaneously redefined as the corresponding v_k. (For more details see [Gri78, GrLe80].)

$SL^S_{S'}$ is statement list SL with every occurrence of statement S replaced by statement S'.

1.5 Thesis Outline

Chapter 2 gives an informal introduction to our methodology and presents a first programming example.

Chapters 3 to 6 build the core of the thesis. They deal with the formal details of the methodology and its programming language RL (for *R*efinement *L*anguage), and demonstrate them on the same programming example. A reader who wants to gain only a quick impression of our research may skip these four chapters and will still understand most of chapters 7 and 8.

Chapter 3 introduces the formal specification of the programming problems the methodology aims to solve. RL programs have to exhibit formal properties which meet the specification. There are two aspects to these properties: semantics (input/output characteristics) and execution time.

Chapter 4 introduces a preliminary proof system for the development of RL programs, the refinement proof system. It is powerful enough to aid the programmer in the discovery of a solution to the semantic part of the problem specification. The refinement proof system describes

(1) the properties of a refinement, namely its semantics and (sequential) execution time, and

(2) semantic properties of parts of the refinement, properties that will suggest computations whose semantics are those of the refinement but whose execution times may be different.

The refinement proof system deals with refinements, not with computations. It can provide an execution time for refinements, but not for computations.

If the refinement's (sequential) execution time is not sufficient, a more detailed proof system, the trace proof system, has to be employed. This is the realm of Chapter 5. The trace proof system describes computations, so-called "traces", as opposed to programs, and serves to verify that there are traces that have the refinement's semantics but are sufficiently fast.

RL programs (described by the refinement proof system) provide algorithmic options. Traces (described by the trace proof system) reflect algorithmic decisions.

Chapter 6 comments on the implementation of RL programs. The central problem is the selection of a satisfactory trace.

The remaining chapters are again less formal.

Chapter 7 presents a collection of further examples. Each makes some point about the methodology. For the readers of chapters 3 to 6, a formal treatment is provided in Appendix A.

Finally, Chapter 8 reviews the flavour of our methodology and relates it to previous work in the area, describes directions for further research, and adds some general comments on the use of programming methodologies.

2 Careful Programming with Concurrency: An Informal Presentation

This chapter gives an informal introduction to the methodology and presents a first programming example.

2.1 Refinement

The first step of the program development is the derivation of a refinement from the input/output assertion pair that specifies the problem.

We use a special language, RL (*R*efinement *L*anguage), for this purpose. RL is closely related to the refinement language of [Heh79]. Its central feature is a primitive procedure concept, which can be implemented efficiently enough to be used extensively as a refinement mechanism.

Statements in RL can either be refined, or basic (not refined). A refined statement is an invented name, say S. Its meaning is conveyed by a refinement, $S: SL$, relating the name S to a refinement body SL. Refinements may be "indexed", e.g., $Sj: SL$, where index j is a variable referenced in SL. An index is a primitive form of value parameter: j may not be changed by SL.

In practice, indices will have to be identified by an index declaration in the refinement, e.g., Sj (j: **int**): SL, where SL refers to j. But, for the sake of brevity, RL does not contain a mechanism for data declarations. We will identify indices as parts of the refinement name that appear in its body.

There are four options of refinement; we call them refinement rules: continuance, replacement, divide (and conquer), and case analysis. Each refinement rule employs a different programming feature: null, assignment, statement composition, or alternation. Divide (and conquer) and case analysis employ a fifth programming feature: the refinement call. Null and assignment are the basic (not refined) statements of RL.

(a) continuance: S: **skip** (null)

skip does nothing at all.

(b) replacement: S: $x:=E$ (assignment)

$x:=E$ gives variable x the value of expression E.

(c) divide-in-2: S: $S1;S2$ (composition)

$S1;S2$ applies statement $S2$ to the results of statement $S1$.

A divide-in-n comprises $n-1$ divide-in-2 in one refinement step. A special case of divide-in-n is the **for** loop. We have a special notation: $\overset{n}{\underset{i=1}{;}} Si$ stands for $S1;...;Sn$.[†]

(d) case analysis: S: **if** $B1 \rightarrow S1 \,\|\, ... \,\|\, Bn \rightarrow Sn$ **fi** (alternation)

The construct $Bi \rightarrow Si$ is called a guarded command [Dij75, Dij76]. Logical expression Bi is guard for alternative Si. The alternative whose guard evaluates to **true** is selected. We restrict case analysis to be deterministic: no two guards of an alternation may be true at the same time.

We write **if** B **then** S **fi** for **if** $B \rightarrow S \,\|\, \sim B \rightarrow$ **skip fi**.

Note the absence of a popular language feature: indefinite repetition. Iterative algorithms are formulated as recursive refinements or, in simple cases, by **for**-composition. While our preference of recursion over repetition is not essential for the methodology, the concept of refinement is.

2.2 Concurrency

The second step of the program development is the declaration of semantic relations between parts of the refinement. Semantic relations may allow relaxations in the sequencing of the refinement's operations. The purpose of their

[†] The loop bounds must be constants in the loop scope. i is a constant for every step Si and local to the loop.

declaration is to speed up the refinement's execution, e.g., by concurrency.

It is important to realize that, while this step may improve the execution time of a solution, it is not going to change or add to its semantics. Semantic declarations only make certain semantic properties, which are already laid down in the refinement, more apparent.

2.2.1 Semantic Relations

Semantic relations between parts of a refinement indicate possible relaxations in sequencing such that its semantics are preserved.

We introduce one unary and four binary relations for refinement components. The unary relation is idempotence; the binary relations are commutativity, full commutativity, non-interference, and independence. Refinement components are either statements or guards. Components in general are denoted with the letter C , statements in particular with S, and guards with B. This section gives only an informal characterization of semantic relations. The precise definitions are in Sect. 4.3.1.

(a) idempotence

$!B$	always	
$!S$	iff	$S;S$ has the same effect as S

An idempotent component C may be applied consecutively any number of times.

(b) commutativity

$B1 \& B2$	always	
$S \& B$	iff	S leaves B invariant
$S1 \& S2$	iff	$S1;S2$ has the same effect as $S2;S1$

Commutative components $C1$ and $C2$ may be applied in any order: $C2$ following $C1$, or vice versa.

(c) full commutativity

C1 ≍ *C2* iff *c1* & *c2* **for all basic components *c1* in *C1* and *c2* in *C2***

Full commutativity is commutativity rippled down the refinement structure. The execution of fully commutative components *C1* and *C2* may be interleaved. Only their basic operations must be indivisible.

(d) non-interference

C1 ⇹ *C2* iff every basic component of *C1* or *C2* gets no more than one view or update of the set of data shared by *C1* and *C2*.

The non-interference of components *C1* and *C2* lifts the mutual exclusions, i.e., permits the divisibility of their basic components (assignment and guard evaluation).

(e) independence

C1 ‖ *C2* iff *C1* ≍ *C2* ∧ *C1* ⇹ *C2*

Independence combines full commutativity with non-interference. Independent components *C1* and *C2* may be executed in parallel (on machines with indivisible memory reference).

Examples:

(1) *S*: *x*:=3, *!S*

(2) *S*: *x*:=*y*, *B*: *x*=3, ~(*S* & *B*), *S* ⇹ *B*

(3) *S1* : *y*:=*x*+1, *S2*: *x*:=3, ~(*S1* & *S2*), *S1* ⇹ *S2*

(4) *S*: *t*:=*p*;*p*:=*q*;*q*:=*t*, *B*: *p*∨*q*, *S* & *B*, ~(*S* ≍ *B*), ~(*S* ⇹ *B*)

(5) *S1* : *t*:=*i*−1;*i*:=*t*;*t*:=*i*+1;*i*:=*t*, *S2*: *j*:=*i*, *S1* & *S2*, ~(*S1* ≍ *S2*), *S1* ⇹ *S2*

(6) *S1* : *x*:=*x*+*a*, *S2*: *x*:=*x*+*b*, *S1* ≍ *S2*, ~(*S1* ‹|› *S2*)

(7) *S*: *c*:=*F*, *B*: *a*∧*b*, *S* ‖ *B*

(8) *S1* : *u*:=*f*(*w*), *S2*: *v*:=*g*(*w*), *S1* ‖ *S2*

Most independence relations will be evident from the following

Independence Theorem:

> Two components *C1* and *C2* of which neither changes any variables appearing in both can be declared independent (proof in Sect. 4.3.1).

All independence relations declared in programming examples of this thesis are applications of the independence theorem.

2.2.2 Semantic Declarations

A semantic relation is declared by stating the relation either after a refinement, or within a refinement by replacing the composition operator ";" with the appropriate relational operator (in-line declaration). Relational operators bind stronger than ";". We will only declare idempotence, commutativity, and independence.

Relations that hold always, such as between guards, do not have to be declared. To this category belong also relations involving **skip**, and relations between S and $\sim B$ if already declared between S and B (see Sect. 4.3.1). Therefore the hidden guarded command $\sim B \rightarrow$ **skip** in **if** B **then** S **fi** can be neglected for semantic declarations. **for** loop index calculations can also be ignored.

A set (or complex) declaration, e.g.

$$\{S1, S2\} \parallel \{T1, T2\}$$

comprises declarations between all set members, in this case,

$$S1 \parallel T1, \quad S1 \parallel T2, \quad S2 \parallel T1, \quad S2 \parallel T2$$

A predicate qualifying the range of refinement index values may be used to define the sets involved. For example,

$$\bigwedge_{i,j} pred(i,j): \quad Si \parallel Sj$$

stands for the set of declarations $Si \| Sj$ such that i and j satisfy $pred(i,j)$. A set index in a complex declaration is passed on to all set members. Assume, for instance, the independence of turn signals of different cars in a traffic system: if we give the operations of the left and right turn signals of car i the names $left_i$ and $right_i$, we will declare

$$\bigwedge_{i,j} i \neq j: \ \{left, right\}_i \| \{left, right\}_j$$

Semantic declarations define additional computations for the refinement they augment. Computations may contain statements and guards with $\rightarrow$ as sequencing operator. We will at this point say no more about the structure of computations and only give a vague idea of the effect of semantic declarations (for details see Chap. 5):

Refinement S is characterized by a set of sequential computations. The semantic declarations extend this set as follows: take some computation for S, then

(a) $!C$ adds computations with instances $C \rightarrow C$ replaced by C, and vice versa,

(b) $C1 \& C2$ adds computations with instances $C1 \rightarrow C2$ or $C2 \rightarrow C1$ swapped,

(c) $C1 \| C2$ adds computations with instances $C1 \rightarrow C2$ or $C2 \rightarrow C1$ replaced by $C1$ parallel with $C2$.

Where further computations can be obtained, the same declarations extend the computation set thus derived, etc. (transitive closure).

Note that semantic relations, although valid, may not be exploitable (i.e., may not generate new computations) for the refinement they are declared for.[†] The declaration and exploitation of semantic declarations are separate concerns. We advise to first declare all semantic relations that can be proved, and only in a later step worry about their exploitability.

[†] Example: A conventional **for** loop implementation with incremental step calculation will render semantic relations between loop steps unexploitable. To exploit them, an index value has to be assigned to every step before any step is executed.

2.3 Example: Sorting

The problem is to sort an array $a[0..n]$ of numbers into ascending order in time $O(n)$. Our refinement is an insertion sort adapted from [KnuIII]:

```
        sort n:  $\overset{n}{\underset{i=1}{;}} S\, i$
        S 0:     skip
(i>0)   S i:     $cs\ i\ ;\ S\ i-1$
        cs i:    if $a[i-1]>a[i]$ then $swap\ i$ fi
        swap i:  $t[i]:=a[i-1];\ a[i-1]:=a[i];\ a[i]:=t[i]$

        $\bigwedge_{i,j} j\neq i-1,i,i+1$:   $cs\ i \parallel cs\ j$
```

Note that for $|i-j|>1$, $cs\ i$ and $cs\ j$ are disjoint: they do not share any variables. Hence they fulfil the premise of the independence theorem and can be declared independent.

To declare semantic relations for some refinement, one does not need to understand the refinement as a whole. A local understanding of the components appearing in the declared relation is sufficient. Most declarations come easily to mind and have a simple proof.

We will not investigate the question of which relations might be automatically declarable, although it is a very interesting one. For the purpose of this thesis, it suffices to assume that the programmer declares every semantic relation; however, we permit the omission of relations that hold always (see Sect. 2.2.2).

3 Problem Specification

3.1 Semantic Specification

The programming methodology presented here can be applied to problems that are described by an assertion pair, what we call a ***semantic specification***. Let us pick a problem and agree on a name for it, say, S. The semantic specification of problem S consists of an input assertion, S. **pre**, and an output assertion, S. **post**:

$$S.\,\mathbf{pre}:\quad P$$
$$S.\,\mathbf{post}:\quad R$$

or, if it is clear that S is the problem referred to,

$$(P,R)$$

where P and R are predicates. P describes the problem's input states and is called the ***input assertion***, R describes the problem's output states and is called the ***output assertion*** of S. The problem name S can be viewed as a statement that has to be refined such as to transform P into R.

We aim for results, and therefore do not permit the false output assertion. This restricts the range of specifiability to finite problems, those which have terminating solutions (the false output assertion indicates non-termination). An infinite problem must be expressed by a specifiable finite segment whose terminating solution can be applied repeatedly.

3.2 Time Specification

The intention is to make solutions to problem S efficient. In general, nothing special has to be specified to express this. But for certain problems an arbitrary attempt may not be good enough. Then the semantic specification is augmented by a ***performance specification.*** We deal only with one aspect of performance: execution time. Other important factors are, for instance, space and number of processors. (Ideally, the sole criterion should be cost, which usually involves all of the above and more.)

Let us assume that result $S.$ **post** is only useful if obtained within a certain time bound. Then a solution to S can only be considered correct if its execution adheres to this time bound. To request a time bound, we add a ***time specification*** to the semantic specification of problem S:

$$\begin{array}{ll} S.\,\textbf{pre}: & P \\ S.\,\textbf{post}: & R \qquad \text{or} \qquad (P,R,t) \\ S.\,\textbf{time}: & t \end{array}$$

t is an integer function $t(\vec{x}_S)$ of the problem's inputs $\vec{x}_S$ ($\vec{x}_S$ is the vector of variables that appear in the input assertion $S.$ **pre**) and defines, for every input, a time bound for the execution of solutions to S. t may also be an "order of" expression.

In order to verify a time bound, the execution time of the operations performed by the used computer hardware has to be known. In this thesis it will be described by a hardware-dependent function, Δ, which maps every hardware operation, say, op on an integer time Δ_{op}. For instance, $\Delta_{:=}$ denotes the execution time of an assignment. By predefining Δ, a time specification may be linked to a special machine, or a class of machines. A requirement that gives weights to operations can be expressed this way. Such requirements are general practice in the analysis of the time complexity of algorithms.

3.3 Example: Sorting

The problem of sorting an array $a[0..n]$ of numbers into ascending order in $O(n)$ time can be specified as follows:

sort n. **pre:** $\bigwedge_{i=0}^{n} a[i] \in \mathbf{R}$

sort n. **post:** $\bigwedge_{i,j}(0 \le i < j \le n \supset a'[i] \le a'[j]) \ \wedge \ \mathrm{perm}(a,a')$

sort n. **time:** $O(n)$

where $\mathrm{perm}(a,a')$ is the predicate that is true if the resulting array a' is a permutation of its original value a, and false otherwise.

4 Program Development (The Refinement Proof System)

In this chapter we introduce a preliminary proof system for the development of RL programs, the ***refinement proof system***. It is powerful enough to aid the programmer in the discovery of a solution to the semantic part of the problem specification. The refinement proof system describes

(1) the properties of a refinement, namely its semantics and (sequential) execution time, and

(2) semantic properties of parts of the refinement, properties that will suggest computations whose semantics are those of the refinement but whose execution times may be different.

The refinement proof system deals with refinements, not with computations. It can provide an execution time for refinements, but not for computations.

If the refinement's (sequential) execution time is not sufficient, a more detailed proof system, the trace proof system, has to be employed. This is the realm of Chapter 5. The trace proof system describes computations, so-called "traces", as opposed to programs, and serves to verify that there are traces that have the refinement's semantics but are sufficiently fast.

RL programs (described by the refinement proof system) provide algorithmic options. Traces (described by the trace proof system) reflect algorithmic decisions.

From now on we will use the following terminology:

Definition:

(a) A ***statement*** is a **skip**, assignment, or refinement call.

(b) A ***refinement list*** is

(i) a statement, or

(ii) a ***composition*** *S1* ; *S2* of two statements *S1* and *S2*, or

(iii) an ***alternation*** **if** $B1 \rightarrow S1$ ⟦ ... ⟦ $Bn \rightarrow Sn$ **fi** using n logical expressions Bi and n statements Si. $Bi \rightarrow Si$ is a ***guarded command*** with ***guard*** Bi and ***alternative*** Si.

(c) A ***refinement*** is the association of a refinement list *SL* with an identifier *S* by *S*: *SL*. We also call *SL* the refinement ***of*** *S*, and *S* the ***refinement name*** for *SL*.

(d) The application of statement *S* in a refinement is an ***instance*** of *S*, or also, if *S* is a refinement name, a ***call*** of *S*.

(e) The statements and guards in the refinement of *S* are the ***primary components*** (***primary statements*** and ***primary guards***) of *S*. The application of a primary component by *S* is a ***primary component instance*** of *S*.

(f) The ***proper components*** of a refinement *S* are its primary components and their proper components (transitive closure).

(g) The ***components*** of *S* are *S* itself and its proper components (reflexive transitive closure).

(h) Statements for which no refinement is given and guards are ***basic***. Statements that are refinement calls are ***refined***.

(j) Variables and constants that appear exclusively in components of *S* are ***local*** to *S*. Variables and constants that appear in *S* and in some *S'* which is not a component of *S* are ***global*** to *S* and ***shared*** by *S* and *S'*. Variables global to *S* that are changed only by components of *S* are ***private*** to *S*.

(k) We will denote a specific instance of component *C* in some refinement list by the lower case equivalent, *c*, and consider this name unique to that instance of *C*. Component instances can be interpreted as different components, each with only a single application (to syntactically transform different instances of component *C* into different components, apply the following replacement algorithm: replace every instance of *C* by a unique name c and add c: *C* before that instance of *C*; if *c* is recursive, identify its recursive calls by c_j: $c^c_{c_{j-1}}$).

(l) Refinement S with a set $\boldsymbol{D}$ of semantic declarations is the ***semantic version*** of S described by $\boldsymbol{D}$, denoted $S\boldsymbol{D}$. (If $\boldsymbol{D}=\phi$, $S\boldsymbol{D}$ is S.)

(m) A semantic version $S\boldsymbol{D}$ that may be called from a user environment is an ***RL program***.

4.1 Timed Assertions

Our methodology will enable not only the determination of total semantic correctness but also the derivation of an upper bound for the program's execution time that might be a prerequisite for performance correctness. Therefore we must use timed assertions for the description of program states. The idea is similar to [Shaw79].

Definition:

A ***timed assertion*** P is of the form $P_{\text{sem}} \wedge P_{\text{time}}$, where the ***semantic part***, P_{sem}, is a predicate about the program's variables, and the ***time part***, or *time stamp*, P_{time}, is a predicate asserting the state of a fictitious variable, *clock*, as an integer function time$_P$ of the vector $\vec{x}$ of program variables:

$$P_{\text{time}} \equiv_{\text{df}} clock \geq \text{time}_P(\vec{x})$$

A timed assertion with time part **true** is a ***semantic assertion***. A timed assertion with semantic part **true** is a ***time assertion*** or ***time stamp***.

Variable *clock* simulates a clock that keeps track of the execution time of the program. It is a ***hidden variable*** [Len78], appearing in assertions but not in programs. It is **not** an auxiliary variable [OwGr76a, OwGr76b]. Both hidden and auxiliary variables need not be implemented, but auxiliary variables must be added to the program to obtain a semantic proof. This methodology does not require auxiliary variables.

The time stamp of timed assertion P specifies an execution time constraint for the program state described by the semantic part of P. In accordance with the weakest precondition calculus in which programs are derived from the postcondition "backwards", we run the program clock backwards, i.e., view time as "running out" rather than progressing. A time stamp may be interpreted as a predicate, P_{time}, or a function time$_P$, whatever is more convenient. To make the parts of timed assertion P explicit, we will occasionally write $(P_{\mathrm{sem}}, P_{\mathrm{time}})$ or $(P_{\mathrm{sem}}, \mathrm{time}_P)$.

Definition:

Consider statement S and assertions P and R. We let $\{P\}\ S\ \{R\}$ stand for an argument that establishes the truth of the formula $P \supset S\{R\}$, i.e., a proof that component S terminates and transforms assertion P into assertion R. We call $\{P\}\ S\ \{R\}$ a proof of total correctness of S with respect to specification $(P_{\mathrm{sem}}, R_{\mathrm{sem}}, \mathrm{time}_P - \mathrm{time}_R)$. The derivation of $\{P\}\ S\ \{R\}$ yields for every statement S' of S an assertion P' satisfying the formula $P' \supset S'\{R'\}$, where R' is an assertion previously derived and known. P' is called the ***precondition***, pre(S'), R' the ***postcondition***, post(S'), of S' in the proof $\{P\}\ S\ \{R\}$.

A proof of S can be outlined by framing every statement in the program text with its pre- and postcondition enclosed in curly brackets. [OwGr76a] calls this a ***proof outline*** and gives an example.

When proving a solution to some timed problem specified by, say, (P,R,t) (where P and R are now **semantic** assertions), we normalize the output time stamp to 0 to obtain as input time stamp t, and write $\{P,t\}\ S\ \{R\}$ for $\{P,t\}\ S\ \{R,0\}$. $\{P,t\}\ S\ \{R\}$ can be read: "in order for S to establish R, nothing more than P has to hold immediately before the execution of S; also, if we start S with a supply of at least t time units ($clock \geq t$), the execution of S will not exceed that supply of time ($clock \geq 0$)." Consequently, the execution of S will not require more than t time units (choose t as input time).

There are, in general, many different proofs of S with respect to (P,R,t). As a solution to a logical inequality, P' is one of many admissible preconditions for statement S' of S. It is best to select the closest possible approximation of weakest precondition $S' \{R'\}$ for the proof. Everything stronger than the weakest precondition adds unnecessary constraints, but the weakest precondition itself may be difficult to express [Dij76].

4.2 Formal Refinement

This section presents the formal definition of the programming language RL and its refinement mechanism. The language features of RL are described by a set of axiomatic weakest preconditions, so-called "language rules". Each language rule defines the semantics and execution time of one programming feature. The process of program refinement is governed by four "refinement rules". Each refinement rule states for a different refinement option under what conditions it is applicable.

Most of this section is condensed from previous publications, notably of Dijkstra [Dij76] and Hehner [Heh79].

4.2.1 Language Rules

This section presents the formal definition of the programming language RL.

In the following rules we denote the evaluation time of expression E by a function $\Upsilon(E)$. This thesis is not concerned with the optimization of expression evaluations (see, e.g., [Kuck77, Sto67] for research in this area), and therefore we do not provide a rigorous definition for $\Upsilon(E)$. However, we will later in this section define $\Upsilon(S)$, the execution time of a statement S.

The rules also refer to an implementation-dependent function Δ that maps each hardware operation on its execution time. A discussion of Δ is beyond the scope of this thesis. Note that for concurrent parts of a computation that are executed on processors of different types different functions Δ apply.

Definition (Language Rules):

For all timed assertions R and some implementation-dependent integer-valued function Δ:

(L1) null: $\mathbf{skip}\{R\} \equiv_{\mathrm{df}} R$

(L2) assignment:

(a) simple: $x:=E\{R\} \equiv_{\mathrm{df}} R^{x,\ clock}_{E,\ clock-\Upsilon(E)-\Delta_{:=}}$

(b) subscripted: $x[E1]:=E2\{R\} \equiv_{\mathrm{df}}$

$$R^{x,\qquad\qquad clock}_{(x;E1:E2),\ clock-\Upsilon(E1)-\Upsilon(E2)-\Delta_{:=}}$$

Thus an array is treated as a (partial) function and an assignment to an array element as a change in the whole function (as in the axiomatic definition of Pascal [HoWi73]).

For the sake of simplicity, we do not distinguish the execution time of a simple and a subscripted memory reference: $\Delta_{:=}$ is identical in both cases. We are also assuming that the value of an expression is always within the domain of the variable it is assigned to.

(L3) composition: $S1;S2\{R\} \equiv_{\mathrm{df}} S1\{S2\{R\}\}$

Note that the time part of this rule reflects the time of sequential execution: the input time stamp of $S2$ serves as output time stamp of $S1$. However, this is only a first estimate. Semantic declarations (Sect. 4.3) may yield computations for $S1;S2$ with improved execution time.

We can parameterize composition: $\mathop{;}\limits_{i=1}^{n} Si$ stands for $S1;\ldots;Sn$. We call this construct a **for** loop. The loop bounds must be constant in the loop scope. i is a constant for every step Si and local to the loop.

(L4) alternation: $\mathbf{if}\ B1 \rightarrow S1\ \|\ \ldots\ \|\ Bn \rightarrow Sn\ \mathbf{fi}\{R\} \equiv_{\mathrm{df}}$

$$\Big(\bigvee_{i=1}^{n}(Bi \wedge Si\{R\})\Big)^{clock}_{clock-\Upsilon(B1,\ldots,Bn)-\Delta_{\mathrm{if}}}$$

$\Upsilon(B1,\ldots,Bn)$ denotes the evaluation time of guards $B1,\ldots,Bn$. We could set $\Upsilon(B1,\ldots,Bn) =_{\mathrm{df}} \Upsilon(B1)+\ldots+\Upsilon(Bn)$, but often not all guards will have to be evaluated. A detailed definition of $\Upsilon(B1,\dots,Bn)$ is not of our concern. Δ_{if} accounts for the branches necessary to select an alternative.

If no guard is true the alternation fails. The present rule assumes that no two guards will be true at the same time, i.e., that the alternation is ***deterministic***. We make this restriction in order to keep operational models for programs simple. Non-determinism requires a backtracking mechanism in trace models for programs [Hoa78a].

(L5) refinement call: (call $S\vec{c}$ with actual indices $\vec{c}$ of refinement $S\vec{y}$: SL with formal indices $\vec{y}$)

(a) no recursion: $S\vec{c}\{R\} \;\equiv_{\text{df}}\; SL\{R\}^{\vec{y},\, clock}_{\vec{c},\, clock-\Upsilon(\vec{c})-\Delta_{\text{call}}}$

Δ_{call} represents the time spent transferring control to the refinement body.

(b) direct recursion:

A recursive refinement is approximated by a sequence of increasingly deeper finite recursions. To express the approximations we need a fail statement that will, however, not appear in RL programs; its sole purpose is to define formally recursive refinement:

fail: **abort**$\{R\} \;\equiv_{\text{df}}\;$ **false**

The ith approximation $(S\vec{y})_i$ of recursive refinement $S\vec{y}$: SL performs at most i recursive steps or fails:

$$(S\vec{y})_0\text{: } \textbf{abort}\,,$$

$$(i>0) \qquad (S\vec{y})_i\text{: } SL^{S\vec{y}}_{(S\vec{y})_{i-1}}\,.$$

Note that $(S\vec{y})_{i-1}$ is a component of $(S\vec{y})_i$. The properties of recursive call $S\vec{c}$ are the limit of the properties of its finite approximations $(S\vec{c})_i$:

$$S\vec{d}\{R\} \equiv_{df} \bigvee_{i\geq 0} (S\vec{d})_i\{R\}$$

(c) indirect or multiple recursion:

The definition of the approximations is messier but conceptually not different: they also represent increasing levels of recursion (see [Heh83]).

To avoid proving a refinement S for different postconditions R, calls can be related to a single refinement proof with respect to, say, postcondition Q (Q should not refer to indices or local variables of S): if $\vec{z}$ is the list of global variables of S and $\vec{u}$ ranges over the values of $\vec{z}$ which establish Q,

$$(S\vec{d}\{Q\} \wedge \bigwedge_{\vec{u}} (Q^{\vec{z}}_{\vec{u}} \supset R^{\vec{z}}_{\vec{u}})) \supset S\vec{d}\{R\}$$

The details of this and a still simpler call rule are discussed in [GrLe80]. A more general reference is [Gri81].

Examples:

(1) $x:=x+1\,\{x=c,\ clock \geq 0\} \equiv (x=c-1,\ clock \geq \Upsilon(x+1)+\Delta_{:=})$

(2) $x:=x+1; x:=x+1\,\{x=c,\ clock \geq 0\} \equiv x:=x+1\{x=c-1,\ clock \geq \Upsilon(x+1)+\Delta_{:=}\}$

$\equiv (x=c-2,\ clock \geq 2\Upsilon(x+1)+2\Delta_{:=})$

(3) **if** $x\neq 0 \rightarrow x:=0 \;[\!]\; x=0 \rightarrow$ **skip fi** $\{x=0,\ clock \geq 0\}$

$\equiv ((x\neq 0,\ clock \geq \Upsilon(0)+\Delta_{:=}) \vee (x=0,\ clock \geq 0))^{clock}_{clock-\Upsilon(x\neq 0,\ x=0)-\Delta_{if}}$

$\equiv^{\dagger} (x\neq 0,\ clock \geq \Upsilon(0)+\Delta_{:=}+\Upsilon(x)+\Delta_{test}+\Delta_{if})$

$\vee\ (x=0,\ clock \geq \Upsilon(x)+\Delta_{test}+\Delta_{if})$

Some remarks on **for** loops are necessary:

A **for** loop is an indexed composition, but the corresponding semantic rule (L3) does not describe the properties of the index calculation. Consequently, although index calculations may be part of a program, they will not be described by that program's formal properties. The following assumptions justify the

† Here we describe the guard evaluation time $\Upsilon(x\neq 0,\ x=0)$ as the time $\Upsilon(x)$ needed to fetch x plus the time Δ_{test} of testing x for 0.

neglect of index calculations:

(a) For any **for** loop, all index values can be calculated before any step is executed.

This assumption permits the neglect of index calculations for semantic declarations.

(b) Index calculations are typically a negligible part of the program.

This assumption justifies the neglect of the execution time of index calculations. (If all index values are calculated concurrently with program parts previous to the **for** loop, their impact on the program's execution time is indeed close to nil.)

The semantic part of (L1) to (L3) is taken from Dijkstra [Dij76], except for the subscripted assignment rule (L2b) which is from [Gri78]. The semantic part of (L4) is a weakened version of Dijkstra's alternation rule [Dij75, Dij76]: we presume deterministic alternations. The semantic part of (L5) is from [Heh79] and subsumes Dijkstra's **do...od** repetition rule [Dij76]. The time part of (L1) to (L5) is new, but a similar execution time calculus for a similar language can be found in [Shaw79].

To test that RL as defined by the language rules has some elementary, always desirable properties, we can check that the language rules satisfy a set of suggested healthiness criteria:

Definition (Healthiness Rules):

(H1) $S\{\mathbf{false}\} \equiv \mathbf{false}$ (excluded miracle)

(H2) $R1 \supset R2 \;\supset\; S\{R1\} \supset S\{R2\}$ (monotonicity)

(H3) $S\{R1\} \wedge S\{R2\} \equiv S\{R1 \wedge R2\}$

(H4) $S\{R1\} \vee S\{R2\} \equiv S\{R1 \vee R2\}$

(H5) $\bigwedge_{k \geq 0} (R_k \supset R_{k+1}) \;\supset\; S\{\bigvee_{k \geq 0} R_k\} \equiv \bigvee_{k \geq 0} S\{R_k\}$ (continuity in postconditions)

(H6) $\bigwedge_{i} (S_i\{R\} \supset S_{i+1}\{R\} \;=\; SL^{S}_{S_i}\{R\} \supset SL^{S}_{S_{i+1}}\{R\})$ (continuity in statements)

(H1) to (H5) have been proposed by Dijkstra [Dij76]; (H6) appeared later. This list is not exhaustive; more healthiness rules could be added. For a proof of

(H1) to (H6) for the semantic part of language rules (L1) to (L5) see [Heh83]. The time part does not pose any healthiness problems: it can in every language rule be interpreted as an assignment to the hidden variable *clock*, and assignments are healthy.

Further properties follow: for instance, we can prove Hoare's "rule of consequence" [Hoa69]:

$$\frac{\{P'\}\ S\ \{R'\},\quad P \supset P',\quad R' \supset R}{\{P\}\ S\ \{R\}}$$

Proof:

We may assume $\{P'\}\ S\ \{R'\},\ \ P \supset P',\ \ R' \supset R$

and have to deduce $\{P\}\ S\ \{R\}$.

$(R' \supset R\,)\ \supset\ (S\{R'\} \supset S\{R\})$ is guaranteed by (H2).

Therefore, using our definition of $\{P'\}\ S\ \{R'\}$,

$\{P'\}\ S\ \{R'\}\ \equiv\ (P' \supset S\{R'\})\ \supset\ (P' \supset S\{R\})$, and thus

$(P \supset P'\,)\ \supset\ (P \supset S\{R\})\ \equiv\ \{P\}\ S\ \{R\}$

The execution time of a statement S is defined as its weakest time precondition with respect to time postcondition 0:

Definition:

The ***execution time*** $\mathrm{T}(S)$ of statement S is $\quad \mathrm{T}(S)\ \equiv_{\mathrm{df}}\ S\{clock \geq 0\}_{\mathrm{time}}$

(Remember that $\mathrm{T}(S)$ can be interpreted as a predicate or a function.)

To conclude this section, let us investigate the execution time of some example refinements:

Consider the following program computing the factorial of n:

```
fact n:  if n=0 → r:=1
         ▯ n>0 → fact n−1; r:=r·n
         fi
```

We can determine the number of multiplications performed by call *fact k* by assuming a machine Δ that only takes time for multiplications (one time unit per multiplication):

$$\begin{array}{lrcl} & \Upsilon(\mathit{fact}\ 0) & = & 0 \\ (k>0) & \Upsilon(\mathit{fact}\ k) & = & \Upsilon(\mathit{fact}\ k-1)+1 \end{array}$$

which yields $\Upsilon(\mathit{fact}\ k)=k$ with domain $k \geq 0$. For the derivation of the formal properties of *fact k* see App. A.1.

To illustrate further that our axiomatic system can be used to determine the time complexity of algorithms, here is a second example, a program which searches for a number x in an ordered array $a[i..j-1]$ (binary search):

```
BS:     if $i=j$ → $found$:=false
        ▯ $i<j$ → $k$:=div($i+j$, 2);
                  if $a[k]<x$ → $i$:=$k+1$; $BS$
                  ▯ $a[k]=x$ → $found$:=true
                  ▯ $a[k]>x$ → $j$:=$k-1$; $BS$
                  fi
        fi
```

We want to count the tests of elements in array $a[i..j-1]$. We therefore set $\Upsilon(a[k]<x,\ a[k]=x,\ a[k]>x)=1$, and let every other operation be for free. We are only interested in the worst case over all possible array inputs of a fixed length, i.e., an upper bound $f(j-i)$ in the array length $j-i$: $\Upsilon(BS) \leq f(j-i)$. One worst case occurs when each comparison of $a[k]$ and x establishes $a[k]<x$. Inspection of BS yields for $f(j-i)$ the recursive equation

$$f(j-i) \;=\; 1 + f(j-\mathbf{div}(i+j,2)+1) \;=\; 1 + f(\lfloor j-\frac{i+j}{2}\rfloor) \;=\; 1 + f(\lfloor\frac{j-i}{2}\rfloor)$$

Following standard complexity methods, we substitute $2^{\log_2(j-i)}$ for $j-i$ to obtain $\Upsilon(BS) \leq f(j-i) = O(\lceil\log_2(j-i)\rceil)$.

4.2.2 Refinement Rules

While the language rules describe the properties of the programming features of RL, the following refinement rules ensure the derivation of a semantically totally correct program, i.e., a program that satisfies the semantic problem specification and provide a proof that complies with the language rules.

Definition (Refinement Rules):

Consider semantic specification (P,R). To obtain a refinement such that $\{P\}\ S\ \{R\}$ choose one of the following:

(R1) continuance: choose S: **skip** if $P \supset R$

(R2) replacement: choose S: $x{:=}E$ if $P \supset R^{x}_{E}$

(R3) divide-in-2: choose S: $S1\,;S2$

if $\bigvee_{Q} (P \supset S1\{Q\} \wedge Q \supset S2\{R\})$

A divide-in-n comprises $n-1$ divide-in-2 refinements in one step. A special case of divide-in-n is the **for** loop (see previous section).

(R4) case analysis: choose S: **if** $B1 \rightarrow S1$ $[\!]$ ... $[\!]$ $Bn \rightarrow Sn$ **fi**

if $P \supset \bigvee_{i=1}^{n} (Bi \wedge Si\{R\})$

Rules (R3) and (R4) ask for further refinements. For more details on their proper choice see the notion of progress in [Heh79].

The refinement rules are taken from [Heh79], except that our case analysis requires deterministic alternations.

4.3 Formal Treatment of Concurrency

The refinement rules guarantee only **semantic** (not time) correctness. A specified execution time constraint may not be met because only semantic assertions, not time stamps, are taken into account. If the refinement is too slow, declarations of semantic relations between certain refinement components have to yield a faster version.

This section deals with the mechanism by which the execution time of refinements can be improved: semantic relations and their declaration. Semantic declarations do not extend or change the refinement in any way. They only make some of its properties more apparent, properties that can be exploited to speed up the execution.

4.3.1 Semantic Relations

Semantic relations provide information about the semantic properties of a refinement. This information can be used to improve the refinement's performance. We consider five semantic relations: idempotence, commutativity, full commutativity, non-interference, and independence.

To determine the full commutativity of two refinements we will, at least in the general case, have to identify the postconditions for their primary component instances. Therefore we introduce the notion of a tail:

Definition:

Consider refinement $S: SL$ and a primary component instance c of S.

The ***tail*** $t(S,c)$ of S with respect to c is the part of SL that succeeds c:

(i) if S: s' then $t(S,s')$: **skip**,

(ii) if S: $s1 ; s2$ then $t(S,s1)$: $s2$, and $t(S,s2)$: **skip**,

(iii) if S: **if** ... ▯ $bi \rightarrow si$ ▯ ... **fi** then $t(S,bi)$: si, and $t(S,si)$: **skip**.

In the following, components in general are denoted with the letter C, statements in particular with S, and guards with B.

As the refinement rules (see previous section), the semantic rules are defined with respect to **semantic** postconditions only:

Definition (Semantic Rules):

A ***semantic relation*** is an expression of the form $!C$, $C1 \,\&\, C2$, $C1 \between C2$, $C1 \nleftrightarrow C2$, or $C1 \,\|\, C2$.

For any semantic assertion R,

(S1) idempotence:

$$!_R\, B \;\equiv_{\mathrm{df}}\; \textbf{true}$$
$$!_R\, S \;\equiv_{\mathrm{df}}\; (\, S\{R\} \equiv S; S\{R\}\,)$$

(S2) commutativity:

$$B1 \,\&_R\, B2 \;\equiv_{\mathrm{df}}\; \textbf{true}$$
$$S \,\&_R\, B \;\equiv_{\mathrm{df}}\; (\, B \wedge S\{R\} \equiv S\{B \wedge R\}\,)$$
$$S1 \,\&_R\, S2 \;\equiv_{\mathrm{df}}\; (\, S1 ; S2\{R\} \equiv S2 ; S1\{R\}\,)$$

(S3) full commutativity:

(a) basic $C1$ and $C2$: $C1 \between_R C2 \;\equiv_{\mathrm{df}}\; C1 \,\&_R\, C2$

(b) basic C, and refined S with primary component instances c_i:

$$S \between_R C \;\equiv_{\mathrm{df}}\; \bigwedge_i c_i \between_{t(S,c_i)\{R\}} C$$

(c) refined S, and refined S' with primary component instances c_i':

$$S \between_R S' \;\equiv_{\mathrm{df}}\; \bigwedge_i S \between_{t(S',c_i')\{R\}} c_i'$$

(S4) non-interference:

$C1 \nleftrightarrow C2 \;\equiv_{\mathrm{df}}\;$ any expression E in $C1$ contains at most one reference to at most one variable changed in $C2$; if $C1$ contains $x{:=}E$ and $C2$ references x, then E does not refer to x nor to any variable changed by $C2$; also, all of the above holds with $C1$ and $C2$ interchanged.

(S5) independence:

$$C1 \,\|_R\, C2 \;\equiv_{\mathrm{df}}\; C1 \between_R C2 \;\wedge\; C1 \nleftrightarrow C2$$

Examples:

(1) $S\colon x{:=}3,\ !_R S \equiv (x{:=}3\{R\} \equiv x{:=}3; x{:=}3\{R\}) \equiv (R^x_3 \equiv (R^x_3)^x_3) \equiv$ **true**

(2) $S\colon x{:=}y,\ B\colon x{=}3,\ S \,\&_R\, B \equiv (x{=}3 \wedge x{:=}y\{R\} \equiv x{:=}y\{x{=}3 \wedge R\}) \equiv$

$(x{=}3 \wedge R^x_y \equiv y{=}3 \wedge R^x_y) \equiv (R^x_y \supset (x{=}3 \equiv y{=}3))$

(3) $S1\colon y{:=}x+1,\ S2\colon x{:=}3,$

$S1 \,\&_{y=c}\, S2 \equiv (y{:=}x+1; x{:=}3\{y{=}c\} \equiv x{:=}3; y{:=}x+1\{y{=}c\}) \equiv$

$(((y{=}c)^x_3)^y_{x+1} \equiv ((y{=}c)^y_{x+1})^x_3) \equiv (x+1{=}c \equiv 3+1{=}c) \equiv (x{=}3)$

Every semantic relation Z_R consists of equivalences $SL1\,\{R\} \equiv SL2\,\{R\}$. In case an equivalence is difficult to prove, try to prove something stronger: the conjunction.

Let us discuss non-interference (S4). This relation is taken from Gries [Gri77], but Gries does not give it a name and calls something else, close to our full commutativity, non-interference. We quote some examples from [Gri77]:

Suppose component *C1* changes a variable a. In order for component *C2* not to interfere with *C1*, it may not contain statements like $a{:=}a+1$ or $b{:=}a+a+1$. If *C1* references a, then in *C2* an assignment $a{:=}a+1$ must be written $t{:=}a+1;\ a{:=}t$, where t is not shared by *C1*. The same restriction holds for an array, where we consider an assignment $a[i]{:=}E$ to be a change of the whole array a. Although the non-interference relation (S4) looks syntactic, it is a semantic condition: if there are subscripts, the set of common variables may depend on the subscripts' values.

(S4) requires indivisibility of memory reference. To quote [Gri77] again:

Suppose component *C1* changes variable (location) A[†] while component *C2* is referencing A. The memory must have the property that the value of A which *C2* receives is the value of A either before or after the update, but not a possible intermediate value.

[†] It is presumed that each assignment updates only one memory location.

There is no reason why we should insist on this specific non-interference criterion (S4) other than that we believe it is the most practical. Other non-interference relations that make different demands at the hardware, e.g., existence of a test-and-set operation may replace (S4). Be aware, however, that the choice of non-interference criterion determines the independence a refinement will contain.

The following relation of ***free*** non-interference does not rely on nice hardware properties. It is stronger than (S4) and thus yields less independence:

(S4') free non-interference:

$C1 \overset{\text{free}}{\nleftrightarrow} C2 \ \equiv_{\text{df}}$ neither $C1$ nor $C2$ reads a bit that the other changes; $C1$ and $C2$ may change a common bit b if all assignments to b by $C1$ or $C2$ yield the same value, and neither $C1$ nor $C2$ reads b.

(S5') free independence:

$$C1 \overset{\text{free}}{\|}_R C2 \ \equiv_{\text{df}} \ C1 \not\asymp_R C2 \ \wedge \ C1 \overset{\text{free}}{\nleftrightarrow} C2$$

Certainly $C1 \nleftrightarrow C2$ does not imply $C1 \overset{\text{free}}{\nleftrightarrow} C2$, but curiously, although it should, $C1 \overset{\text{free}}{\nleftrightarrow} C2$ does not imply $C1 \nleftrightarrow C2$ either. Consider the following situation (due to Eike Best):

Let x and y be two-bit variables, $x,y \in \{0, 1, 2, 3\}$. Then the refinements

$$S1:\ x := 2\cdot(y \ \mathbf{mod}\, 2) + x \ \mathbf{mod}\, 2$$
$$S2:\ x := 2\cdot(x \ \mathbf{mod}\, 2) + y \ \mathbf{mod}\, 2$$

do interfere, $\sim(S1 \nleftrightarrow S2)$, but do not interfere freely, $S1 \overset{\text{free}}{\nleftrightarrow} S2$: $S1$ swaps the high-order bit of x with the low-order bit of y, and $S2$ swaps the other two bits. The reason is that the definition of $\nleftrightarrow$ that we adopted from [Gri77] is, simply but restrictively, phrased in terms of variables, not in terms of bits. Naturally, free non-interference works also on machines with indivisible memory reference.

Proofs of semantic relations follow the same concept as proofs for statements, but the terminology differs somewhat:

Definition:

(1) Consider semantic relation Z and semantic assertions P and R. We let $\{P\}\,Z_R$ stand for an argument that establishes the truth of the formula $P \supset Z_R$. We call $\{P\}\,Z_R$ a ***proof*** of semantic relation Z for ***scope*** (P,R). P is called an ***enabling condition***, R a ***result condition*** for Z.

(2) A proof of semantic relation Z

	for scope	is denoted	and Z is called
(i)	(P,R) for every R	$\{P\}\,Z$	*general under* P
(ii)	$(\mathbf{true},R)$	Z_R	*unconditional for* R
(iii)	$(\mathbf{true},R)$ for every R	Z	*global*

For proofs of full commutativity, $C1 \bowtie C2$, globality is a very important concept. If all mutual commutativities that constitute relation $C1 \bowtie C2$ hold globally, the consideration of intermediate proof assertions in $C1$ and $C2$ can be spared. A full commutativity whose mutual commutativities are not global reflects very difficult semantics for which no easy handles should be expected. In fact, most semantic declarations should be global - at least within the realm of the refinement they are declared for (this weaker form of globality is called "program-specific"; see Sect. 5.3).

As a guideline for a derivation of refinements with potentially high concurrency there is a theorem that guarantees the free independence of two refinement components. Most independence declarations will be applications of this theorem and will not require an extra proof:

Theorem (Independence Theorem):

Two components $C1$ and $C2$ of which neither changes any variables appearing in both can be declared globally freely independent.

Proof:

B1 & B2 :

Always true.

S & B :

Pick any postcondition R. According to the premise B does not refer to program variables changed by S. Therefore S keeps B invariant:

$$S\{B\} \equiv B$$

Commutativity follows by the identities:

$$B \wedge S\{R\} \equiv S\{B\} \wedge S\{R\} \equiv S\{B \wedge R\}$$

S1 & S2 :

Pick any postcondition R. For any statement S, the weakest precondition $S\{R\}$ is derived by substituting variables changed by S in R, maybe using case analysis. According to the premise, the vector $\vec{x}$ of variables changed in one or both of $S1$ and $S2$ is split into two distinct subvectors $\vec{x1}$ of the variables changed by $S1$, and $\vec{x2}$ of the variables changed by $S2$. $S1\{R\}$ results from substituting only variables of $\vec{x1}$ by expressions using only variables of $\vec{x1}$ and constants, and analogously for $S2$. (For the purpose of this proof, program variables changed neither by $S1$ not by $S2$ can be considered constants.) Such distinct substitutions yield the same result, in whatever order performed. Therefore

$$S1\{S2\{R\}\} \equiv S2\{S1\{R\}\}$$

$C1 \mathrel{\times} C2$:

The previous argument can be applied to any pair of components of $C1$ and $C2$.

$C1 \overset{\text{free}}{\not\leftrightarrow} C2$:

Clear: there are no common data.

We can relax the independence theorem in one special case to make full use of the requirement for free non-interference:

Theorem (Supplement to Independence Theorem):

Globally freely independent components *C1* and *C2* may assign the same value to a global bit variable as long as neither of them reads that variable.

We conclude this section with the proof of a property mentioned in Sect. 2.2.2, which permits the neglect of the hidden guarded command $\sim B \rightarrow$ **skip** in **if** B **then** S **fi**:

Lemma:

For all statements S, guards B, and timed assertions R,

$$S \&_R B \quad \supset \quad S \&_R \sim B$$

Proof:

(a) Assume $S \&_R B$ and $R \supset B$.

Then $B \wedge S\{R\} \;\equiv\; S\{B \wedge R\} \;\equiv\; S\{R\}$ yields $S\{R\} \supset B$.

Thus $\sim B \wedge S\{R\} \;\equiv\; \textbf{false} \;\equiv\; S\{\sim B \wedge R\}$, i.e., $S \&_R \sim B$.

(b) Assume $S \&_R B$ and $R \supset \sim B$.

Then $B \wedge S\{R\} \;\equiv\; S\{B \wedge R\} \;\equiv\; S\{\textbf{false}\} \;\equiv\; \textbf{false}$ yields $S\{R\} \supset \sim B$.

Thus $\sim B \wedge S\{R\} \;\equiv\; S\{R\} \;\equiv\; S\{\sim B \wedge R\}$, i.e., $S \&_R \sim B$.

The lemma expresses that statement S keeps guard B invariant if S commutes with B. $S \&_R B$ says that, while establishing R, S preserves the truth of B; $S \&_R \sim B$ says that, while establishing R, S preserves the untruth of B.

4.3.2 Semantic Declarations

Semantic relations are documented in the program text by way of semantic declarations.

Definition:

A ***semantic declaration*** is a semantic relation Z stated after or within some refinement S for some scope, i.e., with optional enabling condition P

and result condition R. For in-line declarations in refinement S, P and R are intermediate proof assertions for S.

The syntax of semantic declarations is described in Sect. 2.2.2. We will only declare idempotence, commutativity, and independence.

4.4 Example: Sorting

We will now prove the sorting program of Sect. 2.3 with respect to the semantic part of the specification in Sect. 3.3. We will also derive the worst-case execution time of the refinement and find out that it does not satisfy the time specification. We do not yet have the tools for a time proof of *sort n* with independence declaration.

Let us denote the execution time of a comparison by *c* and that of a swap by *s*, and let

$$R_{ij} \equiv_{df} R^{a}_{(a;\, i,j:\, a[j],\, a[i])}$$

Then the properties of the refinement's components are:

```
         sort n:   $\mathop{;}\limits_{i=1}^{n} S\,i$
         S0:       $\{R\}$ **skip** $\{R\}$
(i>0)    S i:      $\{cs\,i\,\{S\,i-1\{R\}\}$
                   $cs\,i\,;\,S\,i-1$
                   $\{R\}$
         cs i:     $\{((a[i-1]\leq a[i] \wedge R) \vee$
                        $(a[i-1]>a[i] \wedge swap\,i\,\{R\}_{i-1,i}))^{clock}_{clock-c}\}$
                   **if** $a[i-1]>a[i]$ **then** $swap\;i$ **fi**
                   $\{R\}$
```

$$\textit{swap } i\text{:}\quad \{(R_{i-1,i})^{clock}_{clock-s}\}$$
$$t[i]:=a[i-1];\ a[i-1]:=a[i];\ a[i]:=t[i]$$
$$\{R\}$$

$$\bigwedge_{i,j} j\neq i-1,i,i+1\text{:}\quad cs\ i \parallel cs\ j$$

The recursive expression for $S\,i\{R\}$ will suffice for our purposes. The independence declaration holds globally because its operands obey the independence theorem of Sect. 4.3.1: for $|i-j|>1$, $cs\ i$ and $cs\ j$ do not share any variables.

The properties of *sort n* can be stated precisely but awkwardly.[†] *sort n* $\{R\}$ lists every permutation, its execution time, and the conditions under which it must be applied to sort the array; e.g., for a three-element array (n=2) the following weakest precondition can be formally derived:

$$
\begin{aligned}
\textit{sort } 2\{R\} \quad \equiv\ & (\ (a[0]\le a[1]\le a[2] \wedge R)^{clock}_{clock-3c} \\
\vee\ & (a[0]\le a[2]<a[1] \wedge R_{1,2})^{clock}_{clock-3c-s} \\
\vee\ & (a[2]<a[0]\le a[1] \wedge (R_{0,1})_{1,2})^{clock}_{clock-3c-2s} \\
\vee\ & (a[1]<a[0]\le a[2] \wedge R_{0,1})^{clock}_{clock-3c-s} \\
\vee\ & (a[1]\le a[2]<a[0] \wedge (R_{1,2})_{0,1})^{clock}_{clock-3c-2s} \\
\vee\ & (a[2]<a[1]<a[0] \wedge ((R_{0,1})_{1,2})_{0,1})^{clock}_{clock-3c-3s}\ )
\end{aligned}
$$

The next lemma presents the semantics of *sort n* as a predicate in the variable n.

Let Π_k denote the set of permutations of $(0, 1,\ldots,k)$ with 1_k as identity. Define for any permutation $\pi_k\in\Pi_k$.

[†] The use of permutations is awkward in the inductive assertion method. Hints for specifications which enable easier proofs are in [GeYe76].

$$R_{\pi_k} =_{df} R^{a}_{(a;\,0,\dots,k:\,a[\pi_k(0)],\dots,a[\pi_k(k)])}$$
$$\mathrm{ord}(a[0..k]) =_{df} \bigwedge_{i,j}(0 \le i < j \le k \supset a[i] \le a[j])$$

$R_{1_k} \equiv R$. $\mathrm{ord}(a[0..k])$ says that subarray $a[0..k]$ is in order $\le$. $\mathrm{ord}(a[0..k])_{\pi_k}$ says that $a[0..k]$ is in order $\le$ after permutation π_k.

Lemma:

For semantic R, $\bigwedge_{k \ge 0}(\, sort\ k\,\{R\} \equiv \bigvee_{\pi_k \in \Pi_k} (\mathrm{ord}(a[0..k]) \wedge R)_{\pi_k}\,)$

Proof:

swap i represents the permutation $\varsigma_k^i: (0,\dots,i-1,i,\dots,k) \mapsto (0,\dots,i,i-1,\dots,k)$, $i=1,\dots,k$. Define additionally $\varsigma_k^0 =_{df} 1_k$. Let $\Sigma_k \subseteq \Pi_k$ be the set of permutations σ_k performed by a sequence of strictly decreasing swaps:

$$\Sigma_k =_{df} \{\sigma_k \mid \sigma_k = \varsigma_k^{j_1} \circ \dots \circ \varsigma_k^{j_l},\ 0 \le j_i \le k,\ 1 \le i \le l,\ 1 \le l \le k,\ j_i < j_{i+1}\}$$

Induction on k:

Base: $S\,0\{R\} \equiv R$, $sort\ 0\{R\} \equiv R$

ind. hyp.: $S\,k\{R\} \equiv \bigvee_{\sigma_k \in \Sigma_k} (\mathrm{ord}(a[0..k]) \wedge R)_{\sigma_k}$

$sort\ k\,\{R\} \equiv \bigvee_{\pi_k \in \Pi_k} (\mathrm{ord}(a[0..k]) \wedge R)_{\pi_k}$

ind. step: $S\,k+1\{R\} \equiv cs\ k+1\{S\,k\{R\}\}$

$$\equiv (a[k] \le a[k+1] \wedge S\,k\{R\}) \vee (a[k] > a[k+1] \wedge S\,k\{R\}_{k,k+1})$$
$$\equiv (a[k] \le a[k+1] \wedge \bigvee_{\sigma_k \in \Sigma_k} (\mathrm{ord}(a[0..k]) \wedge R)_{\sigma_k}) \vee$$
$$(a[k] > a[k+1] \wedge (\bigvee_{\sigma_k \in \Sigma_k} (\mathrm{ord}(a[0..k]) \wedge R)_{\sigma_k})_{k,k+1})$$
$$\equiv \bigvee_{\sigma_{k+1} \in \Sigma_{k+1}} (\mathrm{ord}(a[0..k+1]) \wedge R)_{\sigma_{k+1}}$$

$$sort\ k+1\{R\} \equiv sort\ k\,;\,S\,k+1\{R\} \equiv \bigvee_{\pi_k \in \Pi_k} (\mathrm{ord}(a[0..k]) \wedge$$
$$\bigvee_{\sigma_{k+1} \in \Sigma_{k+1}} (\mathrm{ord}(a[0..k+1]) \wedge R)_{\sigma_{k+1}})_{\pi_k}$$
$$\equiv \bigvee_{\pi_{k+1} \in \Pi_{k+1}} (\mathrm{ord}(a[0..k+1]) \wedge R)_{\pi_{k+1}}$$

To explain the last identity: every permutation π_{k+1} can be written as a composition $\sigma_{k+1} \circ \pi_k$ and vice versa.

sort n sorts any $n+1$ elements on which $\leq$ is a total ordering. Thus it satisfies the semantic specification of Sect. 3.3 and sorts any $n+1$ numbers:

$$\begin{aligned}
\textit{sort } n.\,\mathbf{pre} \quad & \supset \bigvee_{\pi_n \in \Pi_n} \mathrm{ord}(a[0..n])_{\pi_n} \\
& \equiv \bigvee_{\pi_n \in \Pi_n} \mathrm{ord}(a[0..n])_{\pi_n} \wedge \mathrm{perm}(a,a) \\
& \equiv \bigvee_{\pi_n \in \Pi_n} (\mathrm{ord}(a[0..n]) \wedge \mathrm{perm}(a,a'))_{\pi_n} \\
& \equiv \bigvee_{\pi_n \in \Pi_n} (\mathrm{ord}(a[0..n]) \wedge \mathrm{ord}(a[0..n]) \wedge \mathrm{perm}(a,a'))_{\pi_n} \\
& \equiv \textit{sort } n \,\{\textit{sort } n.\,\mathbf{post}\}
\end{aligned}$$

We do not have to know the time properties of *sort n* for every array input. An execution time for a worst-case input will do because we specified only a worst-case requirement.

The following lemma provides the worst-case execution time for refinement *sort n*.

Lemma:

$$\bigwedge_{k \geq 0} \Upsilon(\textit{sort } k) \leq \frac{k(k+1)}{2} \cdot c \cdot s$$

Proof:

Induction on k:

Base: $\Upsilon(S\,0) = 0, \;\; \Upsilon(\textit{sort } 0) = 0$

ind. hyp.: $\Upsilon(S\,k) \leq k \cdot c \cdot s\,, \;\; \Upsilon(\textit{sort } k) \leq \frac{k(k+1)}{2} \cdot c \cdot s$

ind. step: $\Upsilon(S\,k+1) \leq \max(1,\, 1+\Upsilon(S\,k)) \leq (1+k) \cdot c \cdot s\,,$

$$\begin{aligned}
\Upsilon(\textit{sort } k+1) &= \Upsilon(\textit{sort } k) + \Upsilon(S\,k+1) \\
&\leq \left(\frac{k(k+1)}{2} + 1 + k\right) \cdot c \cdot s = \frac{(k+1)(k+2)}{2} \cdot c \cdot s
\end{aligned}$$

The assumption that every *cs i* has to swap yields equalities everywhere in the proof: the time bound is exact for an array input in decreasing order.

The refinement alone does not satisfy the time specification: $\frac{k(k+1)}{2} \cdot c \cdot s$ is $O(k^2)$. To verify that the independence declaration decreases the execution time sufficiently we have to use the trace proof system (Chap. 5).

5 The Computations of a Program (The Trace Proof System)

In the previous chapter we have developed a proof system, the refinement proof system, which can describe the properties of refinements and suggest different computations with identical semantics. We now turn to a more powerful proof system, the ***trace proof system***, which can describe these computations. It will serve to formalize the effects of semantic declarations on the set of computations of a refinement.

We represent the computations of a program by sets of directed graphs called traces. (In fact, each trace still represents a set of computations, i.e., is rather a simpler program.) The trace set $\boldsymbol{T}(S)$ of refinement S contains the computations as prescribed by S with composition interpreted as sequential execution. Semantic declarations $D \in \boldsymbol{D}$ for S transform its trace set and any trace set derived by a previous transformation into a semantically equivalent trace set. The transitive closure of all these transformations contains the computations of the semantic version $S\boldsymbol{D}$ of refinement S. The goal is to create a transformation that satisfies not only the semantic but also the time specification of the problem.

5.1 Trace Sets

In this section we will describe the trace set $\boldsymbol{T}(S)$ of a refinement S, i.e., the set of computations as prescribed by S.

Definition:

A ***trace*** τ is a finite directed connected acyclic graph with the following properties:

The node set of τ comprises instances of basic statements and guards in RL, plus ***concurrent command*** nodes, $\langle {\tau 1 \atop \tau 2} \rangle$, where $\tau 1$ and $\tau 2$ are two traces, called the ***tines*** of $<\genfrac{}{}{0pt}{}{\tau 1}{\tau 2}>$. All nodes have indegree 1 and outdegree 1, i.e., a trace is a sequence.

A directed arc between two nodes $v1$ and $v2$, $v1 \rightarrow v2$, is an ***output arc*** of $v1$ and an ***input arc*** of $v2$. An arc is called an ***entry*** (*exit*) ***arc*** of τ if it is not an output (input) arc for any node in τ.

We can compose traces $\tau 1$, $\tau 2$ in sequence to $\tau 1 \rightarrow \tau 2$ by merging the exit arc of $\tau 1$ and the entry arc of $\tau 2$.

Concepts for traces can be applied to trace sets in the usual fashion: for trace sets $\boldsymbol{T1}$, $\boldsymbol{T2}$,

$$\boldsymbol{T1} \rightarrow \boldsymbol{T2} \;=_{\mathrm{df}}\; \{\tau 1 \rightarrow \tau 2 \mid \tau 1 \in \boldsymbol{T1},\ \tau 2 \in \boldsymbol{T2}\}$$

$$<\genfrac{}{}{0pt}{}{\boldsymbol{T1}}{\boldsymbol{T2}}> \;=_{\mathrm{df}}\; \{<\genfrac{}{}{0pt}{}{\tau 1}{\tau 2}> \mid \tau 1 \in \boldsymbol{T1},\ \tau 2 \in \boldsymbol{T2}\}$$

We may sometimes not take the trouble to distinguish a one-element set and its element, and omit the entry and exit arc when spelling out a trace.

The trace set of component C contains the computations as prescribed by C with composition interpreted as sequential execution:

Definition (Computation Rules):

The ***trace set*** $\boldsymbol{T}(C)$ ***of*** component C is of the following form:

(C0) guard: $\boldsymbol{T}(B) =_{\mathrm{df}} \{B\}$

(C1) null: $\boldsymbol{T}(\mathbf{skip}) =_{\mathrm{df}} \{\mathbf{skip}\}$

(C2) assignment:

(a) simple: $\boldsymbol{T}(x:=E) =_{\mathrm{df}} \{x:=E\}$

(b) subscripted: $\boldsymbol{T}(x[E1]:=E2) =_{\mathrm{df}} \{x[E1]:=E2\}$

(C3) composition: $\boldsymbol{T}(S1\,;S2) =_{\mathrm{df}} \boldsymbol{T}(S1) \rightarrow \boldsymbol{T}(S2)$

(C4) alternation: $\boldsymbol{T}(\mathbf{if}\ B1 \rightarrow S1\ \|\ \ldots\ \|\ Bn \rightarrow Sn\ \mathbf{fi}) =_{\mathrm{df}} \bigcup_{i=1}^{n} (\boldsymbol{T}(Bi) \rightarrow \boldsymbol{T}(Si))$

(C5) refinement call: (call $S\vec{c}$ with actual indices $\vec{c}$ of refinement $S\vec{y}$: SL with formal indices $\vec{y}$)

(a) no recursion: $\boldsymbol{T}(S\vec{c}) \;=_{df}\; \boldsymbol{T}(SL)$

(b) (direct) recursion: $\boldsymbol{T}(S\vec{c}) \;=_{df}\; \bigcup_{i\geq 0} \boldsymbol{T}((S\vec{c})_i)$

where $(S\vec{y})_0$: **abort**,

$(i>0)$ $(S\vec{y})_i$: $SL^{S\vec{y}}_{(S\vec{y})_{i-1}}$,

and fail: $\boldsymbol{T}(\textbf{abort}) \;=_{df}\; \phi$

Examples:

(1) $\boldsymbol{T}(\textbf{if } x\neq 0 \rightarrow x:=0 \,[\!]\, x=0 \rightarrow \textbf{skip fi}) = \{x\neq 0 \rightarrow x:=0,\ x=0 \rightarrow \textbf{skip}\}$

(2) $\boldsymbol{T}(fact\ n) = \{n=0 \rightarrow r:=1,$

$n>0 \rightarrow n=0 \rightarrow r:=1 \rightarrow r:=r\cdot n,$

$n>0 \rightarrow n>0 \rightarrow n=0 \rightarrow r:=1 \rightarrow r:=r\cdot n \rightarrow r:=r\cdot n,$

$\ldots\}$

The traces of a refinement S do not reflect its refinement structure: they contain only basic components. Of course, the traces of a refinement do not contain concurrent commands - composition is translated to sequential execution.

Traces do not contain alternations, only guarded commands. The selection of a guarded command from an alternation in RL corresponds to the selection of the trace that contains that guarded command from the alternation's trace set. (Remember that RL programs reflect options while traces reflect decisions.)

The traces of indexed refinements contain formal, not actual indices. The trace set semantics, to be defined next, will attribute to every occurrence of a formal index in a trace the actual index value established by the most recent call.

Let us now describe the properties of the trace set $\boldsymbol{T}(C)$ of component C by trace rules similar to the language rules for RL (see Sect. 4.2.1):

Definition (Trace Rules):

For all timed assertions R and some implementation-dependent integer-valued function Δ:

(T0) guard: $\boldsymbol{T}(B)\{R\} \equiv_{df} B \wedge R^{clock}_{clock-\Upsilon(B)}$

(T1) null: $\boldsymbol{T}(\mathbf{skip})\{R\} \equiv_{df} R$

(T2) assignment:

(a) simple: $\boldsymbol{T}(x{:=}E)\{R\} \equiv_{df} R^{x,\ clock}_{E,\ clock-\Upsilon(E)-\Delta_{:=}}$

(b) subscripted: $\boldsymbol{T}(x[E1]{:=}E2)\{R\} \equiv_{df}$
$R^{x,\qquad clock}_{(x;E1:E2),\ clock-\Upsilon(E1)-\Upsilon(E2)-\Delta_{:=}}$

(T3) composition: $\boldsymbol{T}(S1\,;S2)\{R\} \equiv_{df} \boldsymbol{T}(S1)\{\boldsymbol{T}(S2)\{R\}\}$

(T4) alternation: $\boldsymbol{T}(\mathbf{if}\ B1 \rightarrow S1 \,[\!]\, \ldots \,[\!]\, Bn \rightarrow Sn\ \mathbf{fi})\{R\} \equiv_{df}$
$(\bigvee_{i=1}^{n}(Bi \wedge \boldsymbol{T}(Si)\{R\}))^{clock}_{clock-\Upsilon(B1,\ldots,Bn)-\Delta_{if}}$

(T5) refinement call: (call $S\vec{e}$ with actual indices $\vec{e}$ of refinement $S\vec{y}$: SL with formal indices $\vec{y}$)

(a) no recursion: $\boldsymbol{T}(S\vec{e})\{R\} \equiv_{df} \boldsymbol{T}(SL)\{R\}^{\vec{y},\ clock}_{\vec{e},\ clock-\Upsilon(\vec{e})-\Delta_{call}}$

(b) (direct) recursion: $\boldsymbol{T}(S\vec{e})\{R\} \equiv_{df} \bigvee_{i\geq 0} \boldsymbol{T}((S\vec{e})_i)\{R\}$

where $(S\vec{y})_0$: **abort** ,

$(i>0)$ $(S\vec{y})_i$: $SL^{S\vec{y}}_{(S\vec{y})_{i-1}}$,

and fail: $\boldsymbol{T}(\mathbf{abort})\{R\} \equiv_{df}$ **false**

Definition:

The *execution time* $\Upsilon(\boldsymbol{T}(C))$ of the trace set of component C is

$$\Upsilon(\boldsymbol{T}(C)) \equiv_{df} \boldsymbol{T}(C)\{clock \geq 0\}_{time}$$

The following theorem states that a refinement and its trace set have identical properties. This identity is crucial. It ensures the compatibility of our two proof systems: whatever a refinement represents in the refinement proof system is properly represented by its trace set in the trace proof system.

Theorem:

For all refinements S and timed assertions R, $\boldsymbol{T}(S)\{R\} \equiv S\{R\}$.

Proof:

The identity is directly evident by comparison of language rule (Li) with trace rule (Ti), $i=1,\ldots,5$.

We have defined the trace calculus in terms of trace sets which are derived from refinements. This permitted a trivial transformation of the refinement calculus into a calculus for traces: language rules (L1) to (L5) and trace rules (T1) to (T5) are identical. However, what is really needed is a calculus for traces **independent** of their derivation. Hoare's operational definition of the weakest precondition operator for traces [Hoa78a] provides such a calculus and is consistent with our trace rules.

5.2 Semantic Transformation of Trace Sets

Semantic declarations add computations for S as described informally in Sect. 2.2.2. Formally, a set $\boldsymbol{D}$ of semantic declarations for refinement S transforms the refinement's trace set $\boldsymbol{T}(S)$ or a previous transformation thereof into a new trace set. The transitive closure $\boldsymbol{T}(S\boldsymbol{D})$ of all such transformations contains the computations for the semantic version $S\boldsymbol{D}$ of S.

We will see that all trace sets in $\boldsymbol{T}(S\boldsymbol{D})$ have identical semantics: the semantics of S. We call a trace set with the semantics of S a trace set ***for*** S. $\boldsymbol{T}(S)$ is the unique trace set ***of*** S. We do have a rule for the sequential composition of traces, given by (T3) and (C3). But some of the transformed trace sets will contain concurrency, and we need one more trace rule that describes the properties of concurrent composition:

Definition (Additional Trace Rule):

(T6) concurrent command: $$< \begin{matrix} \boldsymbol{T}(C1) \\ \boldsymbol{T}(C2) \end{matrix} > \{R\} \;\equiv_{\mathrm{df}}\; C1 \,\|_{R_{\mathrm{sem}}}\, C2 \;\wedge$$
$$(\,(\boldsymbol{T}(C1) \to \boldsymbol{T}(C2))\{R_{\mathrm{sem}}\}\,,\ \boldsymbol{T}(C1)\{R_{\mathrm{time}}\} \wedge \boldsymbol{T}(C2)\{R_{\mathrm{time}}\}\,)^{clock}_{clock-\Delta_{<>}}$$

The semantics of a concurrent command are sequential: those of any interleaved computation. The full commutativity included in the independence requirement guarantees that all interleaved computations have the same properties (semantics and execution time).

The execution time of a concurrent command is the maximum of the execution times of its tines, plus some constant $\Delta_{<>}$ that accounts, e.g., for processor management. The non-interference included in the independence requirement guarantees that both tines can be executed in parallel without delays. (We disregard delays due to indivisible memory reference.)

With trace rule (T6) we have the means for expressing parallelism and can turn to the formal description of the effects of semantic declarations. A semantic declaration transforms a trace set by replacing parts of certain traces in it. To identify these parts, we use the concept of a subtrace set:

Definition:

(a) ***T1*** is a ***subtrace set*** of trace set ***T2***,

$$\boldsymbol{T1} \sqsubseteq \boldsymbol{T2} \;\equiv_{\mathrm{df}}\; \bigvee_{\boldsymbol{T}_h, \boldsymbol{T}_t} \boldsymbol{T2} = \boldsymbol{T}_h \to \boldsymbol{T1} \to \boldsymbol{T}_t$$

(b) ***T1*** is a ***subtrace subset*** of trace set ***T2***,

$$\boldsymbol{T1} \sqsubseteq\!\subseteq \boldsymbol{T2} \;\equiv_{\mathrm{df}}\; \bigvee_{\boldsymbol{T}} (\boldsymbol{T1} \sqsubseteq \boldsymbol{T} \wedge \boldsymbol{T} \subseteq \boldsymbol{T2})$$

If $< \begin{matrix} \boldsymbol{T}' \\ \boldsymbol{T}'' \end{matrix} > \sqsubseteq\!\subseteq \boldsymbol{T}$, then $\boldsymbol{T}'$ and $\boldsymbol{T}''$ are called ***tine sets*** of $\boldsymbol{T}$.

Because we transform program components into trace sets we are primarily interested in trace sets, not traces, and subtrace sets, not subtraces - although a subtrace set with only one element can be regarded in place of a subtrace. According to our definition, a collection of subtraces does not necessarily constitute a subtrace set. We require that a subtrace set can be isolated by trace set composition, an operation which in mathematically complex fashion mutually connects all members of the sets involved. Program components can always be isolated this way, and our trace sets are derived from program components. Again, a calculus independent of the derivation from programs could be presented, but it would be more complicated.

The tine sets of trace set $\boldsymbol{T}$ are not subtrace subsets of $\boldsymbol{T}$. In particular, the tines of trace τ are not subtraces of τ. A trace is a **sequence** of nodes. To know all operations of trace τ, we have to look at the nodes of τ and, if we encounter a concurrent command node, look recursively at the nodes of its tines.

Now that we can identify parts of traces and trace sets, let us formalize the substitution of one part for another. Of the following definition, part (a) deals with subtrace subsets, part (b) describes the recursive treatment of concurrent commands:

Definition:

(a) If $\boldsymbol{T}$ is a trace set, $\boldsymbol{T}^{\boldsymbol{T1}}_{\boldsymbol{T2}}$ is $\boldsymbol{T}$ with subtrace (sub)set $\boldsymbol{T1}$ replaced by trace set $\boldsymbol{T2}$,

$$\boldsymbol{T}^{\boldsymbol{T1}}_{\boldsymbol{T2}} \equiv_{\text{df}} \left| \begin{array}{ll} \boldsymbol{T}_{\text{h}} \rightarrow \boldsymbol{T2} \rightarrow \boldsymbol{T}_{\text{t}} & \text{if } \bigvee_{\boldsymbol{T}_{\text{h}}, \boldsymbol{T}_{\text{t}}} \boldsymbol{T} = \boldsymbol{T}_{\text{h}} \rightarrow \boldsymbol{T1} \rightarrow \boldsymbol{T}_{\text{t}} \quad (\text{i.e., } \boldsymbol{T1} \sqsubseteq \boldsymbol{T}) \\ (\boldsymbol{T} \setminus \boldsymbol{T}') \cup \boldsymbol{T}'^{\boldsymbol{T1}}_{\boldsymbol{T2}} & \text{if } \bigvee_{\boldsymbol{T}'} \boldsymbol{T1} \sqsubseteq \boldsymbol{T}' \subseteq \boldsymbol{T} \quad (\text{i.e., } \boldsymbol{T1} \sqsubseteq\subseteq \boldsymbol{T}) \\ \boldsymbol{T} & \text{otherwise} \end{array} \right.$$

For concurrent commands, $\left(<\begin{matrix}\boldsymbol{T}' \\ \boldsymbol{T}''\end{matrix}>\right)^{\boldsymbol{T1}}_{\boldsymbol{T2}} \equiv_{\text{df}} <\begin{matrix}\boldsymbol{T}'^{\boldsymbol{T1}}_{\boldsymbol{T2}} \\ \boldsymbol{T}''^{\boldsymbol{T1}}_{\boldsymbol{T2}}\end{matrix}>$.

(b) Trace set $\boldsymbol{T}'$ is $\boldsymbol{T}$ *with* $\boldsymbol{T1}$ *in place of* $\boldsymbol{T2}$ iff

(i) $\boldsymbol{T}' = \boldsymbol{T}^{\boldsymbol{T1}}_{\boldsymbol{T2}}$, or

(ii) $\boldsymbol{T}_{\text{par}}$ is a concurrent command in $\boldsymbol{T}$, and $\boldsymbol{T}'$ is

$\boldsymbol{T}$ with $(\boldsymbol{T}_{\text{par}})_{\boldsymbol{T2}}^{\boldsymbol{T1}}$ in place of $\boldsymbol{T}_{\text{par}}$.

We will use the notion "$\boldsymbol{T}$ with $\boldsymbol{T1}$ in place of $\boldsymbol{T2}$" to let semantic declarations transform trace sets by replacement of certain of their activities. But first we have to identify which trace set parts ought to be replaced, i.e., we have to select those components in the trace set that match the semantic declaration.

In the subsequent definitions we consider a refinement S, instances $c1$, $c1'$ and $c2$ of components $C1$ and $C2$ of S, a semantic relation Z involving $C1$ and $C2$†, and its declaration D under enabling condition P and for result condition R.

To find out for which parts of trace set $\boldsymbol{T}$ declaration $\boldsymbol{D}$ applies, we proceed in three stages. First, we find all instances of components $C1$ and $C2$ in $\boldsymbol{T}$. Then we select those instances that are adjacent. Only adjacent instances can be candidates for a semantic transformation. Finally we check the scope in which every candidate appears, i.e., inspect the pre- and postcondition of every adjacent instance pair. We may consider performing a transformation only if the scope of the declaration is matched, i.e., if the precondition implies the enabling condition of the declaration and the postcondition is implied by the result condition of the declaration.

Definition:

(1) The ***instance set*** $I_{\boldsymbol{T}}(C)$ of component C in some trace set $\boldsymbol{T}$ is

$$I_{\boldsymbol{T}}(C) \;=_{\text{df}}\; \{\, \text{c} \mid \text{c is an instance of } C\text{, and } \boldsymbol{T}(\text{c}) \sqsubseteq\subseteq \boldsymbol{T} \text{ or } \text{c} \in I_{\boldsymbol{T}'}(C) \text{ for some tine set } \boldsymbol{T}' \text{ of } \boldsymbol{T} \,\}$$

† For idempotence Z involves only $C1$: all definitions apply without consideration of $C2$.

$I_{\boldsymbol{T}}(C)$ is the set of all component instances of C appearing in $\boldsymbol{T}$.

(2) With respect to some trace set $\boldsymbol{T}$, relation Z can be interpreted as

$$Z_{\boldsymbol{T}} =_{df} \{ (c1,c2) \mid \boldsymbol{T}(c1) \rightarrow \boldsymbol{T}(c2) \sqsubseteq \boldsymbol{T} \text{ or } \boldsymbol{T}(c2) \rightarrow \boldsymbol{T}(c1) \sqsubseteq \boldsymbol{T} \text{ or } (c1,c2) \in Z_{\boldsymbol{T}'} \text{ for some tine set } \boldsymbol{T}' \text{ of } \boldsymbol{T}, (c1,c2) \in I_{\boldsymbol{T}}(C1) \times I_{\boldsymbol{T}}(C2) \}$$

$(c1,c2) \in Z_{\boldsymbol{T}}$ is a ***candidate*** for Z in $\boldsymbol{T}$. A candidate is a component pair in $\boldsymbol{T}$ which matches the operands specified in relation Z.

(3) With respect to some trace set $\boldsymbol{T}$ with proof, declaration D can be interpreted as

$$D_{\boldsymbol{T}} =_{df} \{ (c1,c2) \mid \text{pre}(\boldsymbol{T}(c1) \rightarrow \boldsymbol{T}(c2)) \supset P, R \supset \text{post}(\boldsymbol{T}(c1) \rightarrow \boldsymbol{T}(c2)), (c1,c2) \in Z_{\boldsymbol{T}} \}$$

$(c1,c2) \in D_{\boldsymbol{T}}$ is an ***applicator*** of D in $\boldsymbol{T}$. An applicator is a candidate in $\boldsymbol{T}$ which appears in the scope specified in declaration D. D with $C1$, $C2$ replaced by $c1$, $c2$ is the ***declaration instance*** d of D at applicator $(c1,c2)$. An in-line declaration indicates the location of some of its instances.

$D_{\boldsymbol{T}} \subseteq Z_{\boldsymbol{T}} \subseteq I_{\boldsymbol{T}}(C1) \times I_{\boldsymbol{T}}(C2)$.

At last, we are ready to perform a semantic transformation! We can isolate in $\boldsymbol{T}$ an applicator $(c1,c2)$ of declaration $\boldsymbol{D}$ and define the effect of declaration instance d at $(c1,c2)$. d generates from $\boldsymbol{T}$ its transformation $\boldsymbol{T}'$:

Definition (Generation Rules):

The declaration instance d of D at $(c1,c2)$ ***generates*** $\boldsymbol{T}'$ ***from*** $\boldsymbol{T}$ iff

(G1) d is $!c1$, and $\boldsymbol{T}'$ is $\boldsymbol{T}$ with $\boldsymbol{T}(c1)$ in place of $\boldsymbol{T}(c1) \rightarrow \boldsymbol{T}(c1')$, or vice versa, or

(G2) d is $c1 \,\&\, c2$, and $\boldsymbol{T}'$ is $\boldsymbol{T}$ with $\boldsymbol{T}(c2) \rightarrow \boldsymbol{T}(c1)$ in place of $\boldsymbol{T}(c1) \rightarrow \boldsymbol{T}(c2)$, or vice versa, or

(G3) d is $c1 \parallel c2$, and (for arbitrary $\boldsymbol{T}''$) $\boldsymbol{T}'$ is $\boldsymbol{T}$

	with	in place of
either (i)	$< \begin{array}{c} \boldsymbol{T}(c1) \\ \boldsymbol{T}(c2) \end{array} >$	$\boldsymbol{T}(c1) \rightarrow \boldsymbol{T}(c2)$ or $\boldsymbol{T}(c2) \rightarrow \boldsymbol{T}(c1)$
or (ii)	$< \begin{array}{l} \boldsymbol{T}(c1) \rightarrow \boldsymbol{T}'' \\ \boldsymbol{T}(c2) \end{array} >$	$\boldsymbol{T}(c1) \rightarrow < \begin{array}{c} \boldsymbol{T}'' \\ \boldsymbol{T}(c2) \end{array} >$
or (iii)	$< \begin{array}{r} \boldsymbol{T}'' \rightarrow \boldsymbol{T}(c1) \\ \boldsymbol{T}(c2) \end{array} >$	$< \begin{array}{c} \boldsymbol{T}'' \\ \boldsymbol{T}(c2) \end{array} > \rightarrow \boldsymbol{T}(c1)$

or some subsumed commutativity or independence generates $\boldsymbol{T}'$ from $\boldsymbol{T}$.

There must be three rules for the development of concurrency (G3): one to create concurrent commands (i), and two to extend already existing concurrent commands to the right (ii) and to the left (iii).

Definition:

(a) Semantic declaration instance d ***generates*** $\boldsymbol{T}'$ ***from*** $\{ \boldsymbol{T}_i \mid i \in I \}$ iff d generates $\boldsymbol{T}'$ from some $\boldsymbol{T}_i$.

(b) Semantic declaration D ***generates*** $\boldsymbol{T}'$ ***from*** $\boldsymbol{T}$ iff some declaration instance of D generates $\boldsymbol{T}'$ from $\boldsymbol{T}$.

We can now describe the computations of semantic version $S\boldsymbol{D}$. We have to build the transitive closure of all transformations of trace set $\boldsymbol{T}(S)$ by semantic declarations in $\boldsymbol{D}$.

Definition:

The ***transformation set*** $\boldsymbol{T}(S\boldsymbol{D})$ of semantic version $S\boldsymbol{D}$ is defined inductively:

$$\boldsymbol{T}_0(S\boldsymbol{D}) =_{\mathrm{df}} \{ \boldsymbol{T}(S) \}$$

$$(i>0) \quad \boldsymbol{T}_i(S\boldsymbol{D}) =_{\mathrm{df}} \boldsymbol{T}_{i-1}(S\boldsymbol{D}) \cup \{ \boldsymbol{T}_D \mid \bigvee_{D \in \boldsymbol{D}} D \text{ generates } \boldsymbol{T}_D \text{ from } \boldsymbol{T}_{i-1}(S\boldsymbol{D}) \}$$

$$\boldsymbol{T}(S\boldsymbol{D}) =_{\mathrm{df}} \bigcup_{i \geq 0} \boldsymbol{T}_i(S\boldsymbol{D})$$

For all $i>0$, $\boldsymbol{T}_{i-1}(S\boldsymbol{D}) \subseteq \boldsymbol{T}_i(S\boldsymbol{D})$. In the absence of recursion $\boldsymbol{T}(S\boldsymbol{D})$ is finite: there is a k such that $\boldsymbol{T}_{i-1}(S\boldsymbol{D}) = \boldsymbol{T}_i(S\boldsymbol{D})$ for $i \geq k$, indicating that further applications of semantic declarations do not generate new trace sets. If $\boldsymbol{D}=\phi$, then $\boldsymbol{T}(S\boldsymbol{D})=\{\boldsymbol{T}(S)\}$; thus $S\phi$ is S.

We want to prove that the transformation of a trace set does not alter its semantics, i.e., that every trace set in $\boldsymbol{T}(S\boldsymbol{D})$ has the semantics of S. We have, in the refinement proof system, defined semantic relations for refinement components. However, now we are in the trace proof system and are dealing with trace sets, not with refinements. We must characterize semantic relations in terms of trace sets.

The next lemma does just this. It can be viewed as justification for our choice of generation rules. The lemma implies that the transformation of a trace set does not alter its semantics, and the following theorem concludes inductively that all transformations of $\boldsymbol{T}(S)$ must have the semantics of $\boldsymbol{T}(S)$, i.e., the semantics of S.

Lemma:

For any components C, $C1$, $C2$, and semantic assertion R,

(a) $!_R\, C \;\equiv\; (\,\boldsymbol{T}(C)\{R\} \equiv (\boldsymbol{T}(C)\rightarrow\boldsymbol{T}(C))\{R\}\,)$

(b) $C1 \,\&_R\, C2 \equiv (\,(\boldsymbol{T}(C1)\rightarrow\boldsymbol{T}(C2))\{R\} \equiv (\boldsymbol{T}(C2)\rightarrow\boldsymbol{T}(C1))\{R\}\,)$

(For semantic purposes, independence reduces to full commutativity, which is defined in terms of commutativity.)

Proof:

For statements S, $S1$, $S2$, and guards B, $B1$, $B2$, and semantic assertion R,

(a) $!_R\, B \;\equiv\; \textbf{true} \;\equiv\; (\,B\wedge R \equiv B\wedge(B\wedge R)\,)$

$$\equiv\; (\,\boldsymbol{T}(B)\{R\} \equiv (\boldsymbol{T}(B)\rightarrow\boldsymbol{T}(B))\{R\}\,)$$

$!_R\, S \;\equiv\; (\,S\{R\} \equiv S;S\{R\}\,) \;\equiv\; (\,\boldsymbol{T}(S) \equiv (\boldsymbol{T}(S)\rightarrow\boldsymbol{T}(S))\{R\}\,)$

(b) $B1 \,\&_R\, B2 \;\equiv\; \textbf{true} \;\equiv\; (\,B1\wedge(B2\wedge R) \equiv B2\wedge(B1\wedge R)\,)$

$$\equiv\; (\,(\boldsymbol{T}(B1)\rightarrow\boldsymbol{T}(B2))\{R\} \equiv (\boldsymbol{T}(B2)\rightarrow\boldsymbol{T}(B1))\{R\}\,)$$

$S \,\&_R\, B \;\equiv\; (\,B\wedge S\{R\} \equiv S\{B\wedge R\}\,)$

$$\equiv\; (\,(\boldsymbol{T}(B)\rightarrow\boldsymbol{T}(S))\{R\} \equiv (\boldsymbol{T}(S)\rightarrow\boldsymbol{T}(B))\{R\}\,)$$

$S1 \,\&_R\, S2 \;\equiv\; (\,S1;S2\{R\} \equiv S2;S1\{R\}\,)$

$$\equiv\; (\,(\boldsymbol{T}(S1)\rightarrow\boldsymbol{T}(S2))\{R\} \equiv (\boldsymbol{T}(S2)\rightarrow\boldsymbol{T}(S1))\{R\}\,)$$

Theorem:

For any semantic version $\boldsymbol{SD}$ and any semantic assertion R,

$$\bigwedge_{T \in \boldsymbol{T}(\boldsymbol{SD})} \boldsymbol{T}\{R\} \equiv S\{R\}$$

Proof:

Induction on the number of transformations:

Base: $\boldsymbol{D} = \phi$, $\boldsymbol{T}(S\boldsymbol{D}) = \{\boldsymbol{T}(S)\}$, $\boldsymbol{T}(S)\{R\} \equiv S\{R\}$ (previous theorem).

ind. hyp.: Assume an nth transformation $\boldsymbol{T}$ of $\boldsymbol{T}(S)$, and $\boldsymbol{T}\{R\} \equiv S\{R\}$.

ind. step: Let some $d \in \boldsymbol{D}$ generate $\boldsymbol{T}'$ from $\boldsymbol{T}$. Since the weakest precondition of the subtrace set in $\boldsymbol{T}$ which d replaces and its substitute are in all cases identical (previous lemma), $\boldsymbol{T}'$ has the same semantics as $\boldsymbol{T}$. Thus $\boldsymbol{T}'\{R\} \equiv S\{R\}$.

The semantics of different trace sets for S are identical, but their execution times will, in general, differ (otherwise the semantic declarations were pointless). To make the semantic version a solution to the problem, at least one trace set has to stay within the specified time limit:

Definition:

We let $\{P,t\}\ \boldsymbol{SD}\ \{R\}$ stand for an argument that establishes the truth of the formula $(P,t) \supset \boldsymbol{T}\{R, 0\}$ for some trace set $\boldsymbol{T} \in \boldsymbol{T}(\boldsymbol{SD})$, i.e., a proof of total correctness with respect to specification (P,R,t) for some transformation of $\boldsymbol{T}(S)$ by $\boldsymbol{D}$.

The previous theorem allows us to check semantic correctness of $\boldsymbol{SD}$ by proving the refinement S. However, in order to fulfil a time requirement, we may be forced to search $\boldsymbol{T}(\boldsymbol{SD})$ for a suitable transformation. This problem is addressed in Chap. 6.

To conclude this section, let us investigate the computational equivalence of semantic versions:

Definition:

We call two semantic versions $S1\boldsymbol{D1}$ and $S2\boldsymbol{D2}$ of refinements $S1$ and $S2$ ***basically equivalent***, and write $S1\boldsymbol{D1} = S2\boldsymbol{D2}$ iff $S1\boldsymbol{D1}$ and $S2\boldsymbol{D2}$ have the same transformation sets, i.e., $\boldsymbol{T}(S1\boldsymbol{D1}) = \boldsymbol{T}(S2\boldsymbol{D2})$.

The case $\boldsymbol{D1} = \boldsymbol{D2} = \phi$ provides a definition of basic equivalence of refinements.

Basically equivalent refinements must have the same potential for concurrency:

Lemma:

Consider two basically equivalent refinements $S1$ and $S2$.

For every set $\boldsymbol{D1}$ of semantic declarations for $S1$ there is a set $\boldsymbol{D2}$ of semantic declarations for $S2$ such that $S1\boldsymbol{D1} = S2\boldsymbol{D2}$.

Proof:

If $\boldsymbol{D1} = \phi$ set $\boldsymbol{D2} =_{\mathrm{df}} \phi$.

Otherwise, for all declarations $D1 \in \boldsymbol{D1}$, $D1$ can be replaced by the sequence of all declaration instances $d1$ of $D1$ in $\boldsymbol{T}(S1\boldsymbol{D1})$, yielding a declaration set $\boldsymbol{D1}'$ such that $S1\boldsymbol{D1} = S1\boldsymbol{D1}'$. (Joining the declaration instances at calls of a recursion into a set declaration keeps $\boldsymbol{D1}'$ finite.) By expanding refinement calls we can transform $\boldsymbol{D1}'$ into a set $\boldsymbol{D1}''$ of declarations each of which only involves concrete components; $S1\boldsymbol{D1}'' = S1\boldsymbol{D1}'$. Since basically equivalent refinements have identical trace sets, $\boldsymbol{D1}''$ has identical effects on $S1$ and $S2$. Set $\boldsymbol{D2} =_{\mathrm{df}} \boldsymbol{D1}''$.

Although $S1\boldsymbol{D1}$ and $S2\boldsymbol{D2}$ may be basically equivalent, one may be preferable to the other: the conciseness with which the semantic relations in a refinement can be described depends on the refinement structure. There may also be refinements basically different from $S1$ and $S2$ that satisfy the semantic specification and have better concurrency properties.

The trace proof system can describe the properties of a semantic version $\boldsymbol{SD}$. It can tell whether $\boldsymbol{SD}$ is a solution to the specified problem, but not whether it is an optimal solution or what an optimal solution will look like. Our methodology emphasizes the development of safe, not of optimal programs.

5.3 Program-Specific Semantic Declarations

The purpose of enabling and result conditions in semantic declarations is to determine the applicators of a declaration among the candidates for the declared relation. However, it may be that the enabling or result condition of some declaration is satisfied by all candidates and therefore irrelevant for the program in which the declaration appears. An obviously irrelevant condition may be omitted, rendering the declaration ***program-specific*** - applicable only to that program in which the condition is superfluous.

Definition:

(1) Consider a trace set $\boldsymbol{T}$ and a semantic relation Z with scope (P',R'). If $\{P\}\ \boldsymbol{T}\ \{R\}$ is a proof with respect to semantic specification (P,R), define

(i) $POST(P')$ as the conjunction of all postconditions of candidates for Z in $\boldsymbol{T}$ which have a precondition implying P',

(ii) $PRE(R')$ as the disjunction of all preconditions of candidates for Z in $\boldsymbol{T}$ which have a postcondition implied by R'.

(a) Z is general under P' ***in*** $\boldsymbol{T}$ with respect to (P,R) iff there is a proof $\{P\}\ \boldsymbol{T}\ \{R\}$ such that $R' \supset POST(P')$.

(b) Z is unconditional for R' ***in*** $\boldsymbol{T}$ with respect to (P,R) iff there is a proof $\{P\}\ \boldsymbol{T}\ \{R\}$ such that $PRE(R') \supset P'$.

(c) Z is global ***in*** $\boldsymbol{T}$ with respect to (P,R) iff there is a proof $\{P\}\ \boldsymbol{T}\ \{R\}$ such that $R' \supset POST(\mathbf{true})$ and $PRE(\mathbf{false}) \supset P'$.

(2) Consider a semantic version $\boldsymbol{SD}$ where $\boldsymbol{D}$ declares relation Z for scope (P',R').

Z is general under P' (unconditional for R', global) ***in SD*** iff
Z is general under P' (unconditional for R', global) in every $\boldsymbol{T} \in \boldsymbol{T}(\boldsymbol{SD})$.

Relations that hold generally (unconditionally, globally) in ***SD*** may be declared like general (unconditional, global) relations, i.e., in declarations

(i) the result condition may be omitted if Z is general in ***SD***,

(ii) the enabling condition may be omitted if Z is unconditional in ***SD***,

(iii) the enabling and result conditions may be omitted if Z is global in ***SD***.

If the generality (unconditionality, globality) of a relation in ***SD*** can easily be detected much effort may be saved: the checking of intermediate assertions in the traces in order to identify certain candidates as applicators becomes unnecessary. For a semantic relation global in ***SD*** **every** candidate is an applicator.

5.4 Example: Sorting

We have to look at the transformation set of the semantic version of *sort n* presented in Sect. 2.3 and investigate if some transformation of the refinement trace set $\boldsymbol{T}(sort\ n)$ satisfies the time limit of $O(n)$ specified in Sect. 3.3.

Our trace notation has been designed to describe the effects of semantic declarations, not to describe specific traces or trace sets. We present some useful new notation now, but leave a more involved syntax for concise trace proofs to the user:

$$\overset{n}{\underset{i=1}{\rightarrow}} \tau_i =_{\mathrm{df}} \tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n$$

$$\langle \tau_i \rangle_{i=1}^{n} =_{\mathrm{df}} \begin{cases} \tau_1 & \text{if } n=1 \\ \left\langle \begin{matrix} \langle \tau_i \rangle_{i=1}^{n-1} \\ \tau_n \end{matrix} \right\rangle & \text{if } n>1 \end{cases}$$

The computation rules transform the refinement *sort n* into the following trace set:

$$\boldsymbol{T}(sort\ n) \;=\; \overset{n}{\underset{i=1}{\rightarrow}} (\overset{i-1}{\underset{j=0}{\rightarrow}} \boldsymbol{T}(cs\ i-j))$$

We may view *cs i* as a basic statement because the independence declaration for *sort n* does not refer to proper components of *cs i*. This allows us to interpret trace sets for *sort n* as single traces:

$$\boldsymbol{T}(sort\ n) \;=\; \tau_n \;=_{df}\; \overset{n}{\underset{i=1}{\rightarrow}} (\overset{i-1}{\underset{j=0}{\rightarrow}} cs\ i-j)$$

e.g., for a five-element array ($n=4$),

$$\tau_4 \;=\; cs\ 1 \rightarrow cs\ 2 \rightarrow cs\ 1 \rightarrow cs\ 3 \rightarrow cs\ 2 \rightarrow cs\ 1 \rightarrow cs\ 4 \rightarrow cs\ 3 \rightarrow cs\ 2 \rightarrow cs\ 1$$

We proved in Sect. 4.4 that the worst-case execution time of τ_n is $O(n^2)$.

We will now prove that τ_n can be transformed by the given independence declaration to

$$\tilde{\tau}_n \;=_{df}\; (\overset{n-1}{\underset{i=1}{\rightarrow}} v_i) \rightarrow (\overset{n-1}{\underset{i=0}{\rightarrow}} v_{n-i})$$

$$\text{where}\quad v_i \;=_{df}\; <cs\ i-2j>_{j=0}^{\frac{i-1}{2}}$$

e.g., for a five-element array,

$$\tilde{\tau}_4 \;=\; cs\ 1 \rightarrow cs\ 2 \rightarrow <\begin{matrix} cs\ 3 \\ cs\ 1 \end{matrix}> \rightarrow <\begin{matrix} cs\ 2 \\ cs\ 4 \end{matrix}> \rightarrow <\begin{matrix} cs\ 1 \\ cs\ 3 \end{matrix}> \rightarrow cs\ 2 \rightarrow cs\ 1$$

If we neglect processor management ($\Delta_{<>}=0$[†]), $\tilde{\tau}_n$ has a worst-case execution time of $(2n-1)\cdot c\cdot s$, i.e., $O(n)$. $\tilde{\tau}_n$ is fastest for this semantic version of *sort n*, but we need not prove that. We only have to know that it satisfies the specified time limit.

[†] To obtain linear execution time for a program with unbounded concurrency, $\Delta_{<>}$ must be negligible.

Lemma:

For all n, τ_n can be transformed into $\tilde{\tau}_n$.

Proof:

Induction on n:

Base: $\tau_1 = \tilde{\tau}_1 = cs\ 1$ (identity transformation).

ind. hyp.: τ_n can be transformed to $\tilde{\tau}_n$.

ind. step: $\tau_{n+1} = \overset{n+1}{\underset{i=1}{\rightarrow}} (\overset{i-1}{\underset{j=0}{\rightarrow}} cs\ i-j) = \overset{n}{\underset{i=1}{\rightarrow}} (\overset{i-1}{\underset{j=0}{\rightarrow}} cs\ i-j) \rightarrow (\overset{n}{\underset{j=0}{\rightarrow}} cs\ i-j)$

can by ind. hyp. be transformed to

$$\tau'_{n+1} =_{df} (\overset{n-1}{\underset{i=1}{\rightarrow}} < cs\ i-2j >_{j=0}^{\frac{i-1}{2}}) \rightarrow$$

$$(\overset{n-1}{\underset{i=0}{\rightarrow}} < cs\ n-i+2j >_{j=0}^{\frac{n-i-1}{2}}) \rightarrow (\overset{n}{\underset{j=0}{\rightarrow}} cs\ n+1-j)$$

$$= (\overset{n}{\underset{i=1}{\rightarrow}} < cs\ i-2j >_{j=0}^{\frac{i-1}{2}}) \rightarrow$$

$$(\overset{n-1}{\underset{i=1}{\rightarrow}} < cs\ n-i+2j >_{j=0}^{\frac{n-i-1}{2}}) \rightarrow (\overset{n}{\underset{j=0}{\rightarrow}} cs\ n+1-j)$$

The inductive step is to show that

$$\lambda_{n+1} =_{df} (\overset{n-1}{\underset{i=1}{\rightarrow}} < cs\ n-i+2j >_{j=0}^{\frac{n-i-1}{2}}) \rightarrow (\overset{n}{\underset{j=0}{\rightarrow}} cs\ n+1-j)$$

transforms to

$$\tilde{\lambda}_{n+1} =_{df} (\overset{n-1}{\underset{i=1}{\rightarrow}} < cs\ n+2-i+2j >_{j=0}^{\frac{n-i-1}{2}}) \rightarrow cs\ 2 \rightarrow cs\ 1$$

$$= (\overset{n-2}{\underset{i=1}{\rightarrow}} < cs\ n+1-i+2j >_{j=0}^{\frac{n-i}{2}}) \rightarrow$$

$$(\overset{n}{\underset{i=n-1}{\rightarrow}} < cs\ n+1-i+2j >_{j=0}^{\frac{n-i}{2}})$$

$$= \overset{n}{\underset{i=0}{\rightarrow}} < cs\ n+1-i+2j >_{j=0}^{\frac{n-i}{2}}$$

Then τ'_{n+1} transforms to

$$\tau''_{n+1} =_{\text{df}} (\overset{n}{\underset{i=1}{\rightarrow}} < cs\ i-2j >_{j=0}^{\frac{i-1}{2}}) \rightarrow (\overset{n}{\underset{i=0}{\rightarrow}} < cs\ n+1-i+2j >_{j=0}^{\frac{n-i}{2}})$$

$$= \tilde{\tau}_{n+1}$$

To show the inductive step, let us define

$$\kappa_{n+1} =_{\text{df}} \overset{n-1}{\underset{i=1}{\rightarrow}} < cs\ n-i+2j >_{j=0}^{\frac{n-i-1}{2}}$$

$$\mu_{n+1} =_{\text{df}} \overset{n}{\underset{j=0}{\rightarrow}} cs\ n+1-j$$

$$\tilde{\kappa}_{n+1} =_{\text{df}} \overset{n-1}{\underset{i=1}{\rightarrow}} < cs\ n+2-i+2j >_{j=0}^{\frac{n-i-1}{2}}$$

Then $\lambda_{n+1} = \kappa_{n+1} \rightarrow \mu_{n+1}$, $\tilde{\lambda}_{n+1} = \tilde{\kappa}_{n+1} \rightarrow cs\ 2 \rightarrow cs\ 1$.

Remember that $cs\ i \parallel cs\ j$ for $|i-j|>1$. With $k=1,\ldots,n-1$, the highest index in the kth node (concurrent command) of κ_{n+1} is $n-k$, the index of the kth node of μ_{n+1} is $n-k+2$. Note that the indices in both κ_{n+1} and μ_{n+1} are monotonically decreasing, i.e., low indices in κ_{n+1} meet high indices in μ_{n+1}. Thus the kth node of μ_{n+1} can be commuted to the kth concurrent command of κ_{n+1} (G2) and ravelled up as a new tine (G3i). Doing this for all k, we obtain $\tilde{\kappa}_{n+1}$. μ_{n+1} has two more nodes, $cs\ 2 \rightarrow cs\ 1$, which cannot be ravelled. This leaves us with $\tilde{\lambda}_{n+1}$.

$\tilde{\tau}_n$ sorts array $a[0..n]$ in $2n-1$ steps. $a[0..n]$ can be sorted faster, in approximately n steps, by alternately comparing all odd pairs in parallel and all even pairs in parallel, $n/2$ times. However, the according trace

$$\overset{\lceil \frac{n}{2} \rceil}{\underset{1}{\rightarrow}} (v_{\text{odd}} \rightarrow v_{\text{even}})$$

$$\text{where} \quad v_{\text{odd}} =_{\text{df}} < cs\ 2j-1 >_{j=1}^{\lceil \frac{n}{2} \rceil},$$

$$v_{\text{even}} =_{\text{df}} < cs\ 2j >_{j=1}^{\frac{n}{2}}$$

cannot be derived from our refinement of *sort n*, not even if we additionally declare the idempotence of *cs i*, for all *i*. The trace does not constitute a bubble sort. For computations which are not a bubble sort, *sort n* must be refined differently.

5.5 Operational Models for Traces

We have already in Sect. 5.1 referred to an operational representation of traces: the exhaustive study of Hoare [Hoa78a]. Although Hoare considers only sequential execution, the semantics of concurrent command rule (T6) are clearly consistent with the arbitrary interleaved execution of the two tines in his operational model: the semantic part of (T6) is the weakest precondition of every interleaved execution.

Operational models commonly simulate concurrency by interleaved execution, but if we want to model execution time and not only semantics we are in need of truly parallel execution. We can represent it by tokens propagating along the arcs of traces.

Node v of trace τ is activated when the input arc of v carries a token. Upon termination of the node, the token is transferred from the input to the output arc of v. If a guard evaluates to **false**, a token is placed on an imaginary second output arc instead. We call it ***abort* arc** since a token on it signifies abortion of the trace. Abort arcs are exit arcs. Activating a concurrent command means placing a token on the entry arc of every tine.

A call of a refinement S places a token on the entry arc of every trace in its trace set $\boldsymbol{T}(S)$. If a token reaches the exit arc of its trace, that trace models a legal computation for this call. All traces are simultaneously executed to the point of termination or abortion.

The call of a refinement S with semantic declarations $\boldsymbol{D}$ can be modelled by simultaneous calls of all trace sets $\boldsymbol{T} \in \boldsymbol{T}(S\boldsymbol{D})$. The transformations whose tokens come to rest first are fastest.

We can use a formalism to describe the safe movement of tokens as defined by Lamport [Lam77] for a set of N concurrent processes built from nodes of two types: assignment and decision. Execution is modelled by the transformation of program states that are values of variables paired with arcs in the process graph. Assertions labelling arcs describe the legal states.

We replace Lamport's processes by traces and consider only the cases $N=1$ modelling sequential execution and $N=2$ modelling binary concurrency. Higher than binary concurrency can be simulated by nested binary concurrent commands.

To be able to use Lamport's formalism, we have to classify trace nodes as assignments and decisions:

(a) basic statements (null and assignment) and concurrent commands are assignment nodes,

(b) guards are decision nodes (remember the added abort arcs).

Lamport calls a mapping of an assertion on each arc of trace τ an ***interpretation*** I of τ. I is ***consistent at node*** v iff $P \supset v\{R\}$, where P labels the input arc and R the output arc of v. I is ***consistent*** iff it is consistent at every node of τ. Any set of intermediate assertions for trace τ derived in conformity with trace rules (T0) to (T6) is consistent.

Lamport calls an interpretation that is a proof for τ with respect to some semantic specification (P,R) ***invariant*** with respect to P. To assure the concurrent correctness of two tines of a concurrent command, they must have invariant interpretations that are mutually ***monotone***, Lamport's term for consistency on shared data. Our notion of independence implies Lamport's monotonicity.

We encourage the reader to study Lamport's model [Lam77] (the formalism fills only one page). Lamport does not consider execution time, but an extension is simple: include the effect of nodes in the hidden variable *clock* (this converts decision nodes into a second type of assignment nodes), but disregard *clock* in monotonicity arguments.

6 Implementation

In the previous chapter we have described a semantic version ***SD*** by the transformation set $\boldsymbol{T}(\boldsymbol{SD})$. Its members are the trace set $\boldsymbol{T}(S)$ of refinement S and a number of transformed trace sets with the same semantics as $\boldsymbol{T}(S)$. This chapter discusses the implementation of ***SD***, i.e., the selection of a suitable trace from $\boldsymbol{T}(\boldsymbol{SD})$ for execution.

First some general remarks:

Executing a program concurrently is not going to be simpler or cheaper than a sequential execution. It is going to be faster, though, and in order to save time in execution we will, in general, have to expend additional effort previously.

Let us assume a program that we cannot or do not want to optimize. However, we want it to take less time than it does if executed in sequence. To keep the length of the execution within the desired limits, we have to ravel the program's components into a sufficient number of concurrent tines. This is no improvement of the program's computing effort. We merely choose to invest in processing power rather than execution time.

The following observation tells us the range of execution speed-up we can expect from a semantic transformation:

Observation:[†]

Let the ***degree of concurrency***, or ***width***, $\Gamma(\boldsymbol{T})$ of trace set $\boldsymbol{T}$ be the maximum number of parallel tines in any trace of $\boldsymbol{T}$. $\Upsilon(\boldsymbol{T})$ is the execution time of $\boldsymbol{T}$ as defined in Sect. 5.1.

Then, for every trace set $\boldsymbol{T} \in \boldsymbol{T}(\boldsymbol{SD})$ generated from trace set $\boldsymbol{T}(S)$ of some refinement S by semantic declarations D (without idempotence),

[†] This is really an observation about individual traces, not trace sets. But we are here primarily concerned with trace sets.

$$\frac{\Upsilon(\boldsymbol{T}(S))}{\Gamma(\boldsymbol{T})} \leq \Upsilon(\boldsymbol{T}) \leq \Upsilon(\boldsymbol{T}(S))$$

The best we can hope for is that $\boldsymbol{T}$ maintains its maximum degree of concurrency all the way, dividing the length of $\boldsymbol{T}(S)$ by $\Gamma(\boldsymbol{T})$. In other words, the execution ***speed-up*** of the refinement by trace set $\boldsymbol{T}$ is bounded by the degree of concurrency of $\boldsymbol{T}$:

$$\frac{\Upsilon(\boldsymbol{T}(S))}{\Upsilon(\boldsymbol{T})} \leq \Gamma(\boldsymbol{T})$$

Phrasing it more loosely (neglecting constants): to save some order of execution time, we have to add that same order of concurrency. Take, for example, the sorting program *sort n*. Its sequential execution time is $O(n^2)$ (see Sect. 4.4). A concurrency degree of $O(n)$ improves the execution time to $O(n)$ (see Sect. 5.4).

However, the order of execution speed-up may fall short of the order of number of processors involved in the execution, unless concurrency is maintained to an adequate extent throughout the execution.

The derivation of the concurrent from the sequential trace set embodies the additional effort we have to expend. This is part of the concern of this chapter.

We also have to describe the execution of the concurrent trace set we arrive at. In the operational models (Sect. 5.5) we view the execution of a trace set as the simultaneous execution of all its traces. For an implementation, we have to be more realistic. There is an easy implementation for any refinement S, and therefore also for its trace set $\boldsymbol{T}(S)$. But we will have to restrict the effect of semantic declarations in order to keep transformed trace sets as easily implementable.

6.1 Trace Sets

This section discusses the selection of a trace from some trace set $\boldsymbol{T}$ for refinement S for execution.

For $\boldsymbol{T}(S)$, the trace set of S, a selection is easy: if S does not contain alternatives, there is only one trace, and for every alternative IF in S, the trace of $\boldsymbol{T}(IF)$ to be executed can be chosen by evaluating the guards of IF in any order.

Guard evaluation as a mechanism for trace selection works only if the traces containing some guard of IF are indistinguishable up to that guard. In other words, trace differences must always originate at guards. We will call a trace set with this property ***simple.*** It can be represented by a tree whose branches originate in guards. Each path form the root to a leaf of the tree corresponds to one trace in the set.

For every refinement S, $\boldsymbol{T}(S)$ is simple. However, semantic transformations may introduce non-simple trace sets. Consider the following situation:

S: $S0\,;IF$

IF: **if** $B1 \rightarrow S1 \,[\!]\, B2 \rightarrow S2$ **fi**

$S0 \,\&\, B1\ ,\ S0 \,\&\, B2$

The trace sets for S in this semantic version are

$$\begin{aligned}
\boldsymbol{T1} &=_{\mathrm{df}} \{S0 \rightarrow B1 \rightarrow S1\ ,\ S0 \rightarrow B2 \rightarrow S2\} = \boldsymbol{T}(S)\\
\boldsymbol{T2} &=_{\mathrm{df}} \{B1 \rightarrow S0 \rightarrow S1\ ,\ S0 \rightarrow B2 \rightarrow S2\}\\
\boldsymbol{T3} &=_{\mathrm{df}} \{S0 \rightarrow B1 \rightarrow S1\ ,\ B2 \rightarrow S0 \rightarrow B2\}\\
\boldsymbol{T4} &=_{\mathrm{df}} \{B1 \rightarrow S0 \rightarrow S1\ ,\ B2 \rightarrow S0 \rightarrow B2\}
\end{aligned}$$

$\boldsymbol{T1}$ and $\boldsymbol{T4}$ are simple, but $\boldsymbol{T2}$ and $\boldsymbol{T3}$ are not. We have to restrict the effect of semantic declarations such that only simple trace sets can be generated.

Restriction I:

Let $\circ$ stand for any binary semantic relation. Consider some component C, and guarded command $Bk \rightarrow Sk$ of some alternation IF with n guarded commands. Consider a semantic declaration $Zi =_{\mathrm{df}} C \circ Bi$ $(i=1,\ldots,n)$.

Then we may apply **all** Zi **simultaneously** to some instance of IF, but if Zi does not hold for some i, no Zj $(j \neq i)$ may take effect.

With this restriction, all guards of an alternative are subjected to identical semantic transformations and simpleness is preserved.

If we apply it to the example, ***T2*** and ***T3*** are eliminated. And withdrawing one of the semantic declarations has the effect of withdrawing both.

6.2 Semantic Transformation of Trace Sets

This section discusses the selection of a trace set $\boldsymbol{T}$ for refinement S from the transformation set $\boldsymbol{T}(\boldsymbol{SD})$ of semantic version $\boldsymbol{SD}$ for execution.

We will perform most of the trace set transformations before run time. An execution in the specified time limit must be guaranteed before run time. But we will also during execution allow final touches that require no or negligible additional processing.

6.2.1 Before Run Time

If we represent the trace set $\boldsymbol{T}(S)$ of refinement S as the computation rules (C0) to (C5) suggest (Sect. 5.1), $\boldsymbol{T}(S)$ can be computed in time linear with respect to the length of S.

The transformation set $\boldsymbol{T}(\boldsymbol{SD})$ of a semantic version $\boldsymbol{SD}$ of refinement S is recursively enumerable (Sect. 5.2). However, the length of the transformation sequence that generates some $\boldsymbol{T}$ from $\boldsymbol{T}(S)$ may be immense - as bad as the sequential execution of S for every input.

The complexity of transformation of $\boldsymbol{T}(S)$ is governed by two factors:

(1) The number of traces (the cardinality of $\boldsymbol{T}(S)$).

The cardinality of $T(S)$ grows exponentially with the number of alternations in S: if S has m alternations with n guards each, $\boldsymbol{T}(S)$ contains n^m traces.

(2) The number of applicators in the traces.

The number of transformations of a trace grows exponentially with the number of applicators in that trace: n applicators generate 2^n-1 traces, one for each combination of applications.

Here is some advice on how the complexity of semantic transformations can be reduced:

(1) To mitigate the impact of a large number of traces, avoid declarations that refer to components inside alternatives. For the purpose of semantic transformations, alternations that are below the level of semantic declarations can be viewed as basic statements: their alternatives need not be individually transformed.

In Sect. 6.3 we mention a class of sorting programs whose declarations never need to refer inside alternatives. For the purpose of semantic declarations, their trace sets reduce to single traces.

(2) Compilable concurrency must be the result of finitely many semantic transformations. The following restriction cuts $\boldsymbol{T}(S\boldsymbol{D})$ down to finite cardinality:

Restriction II:

A semantic transformation inside a recursion may only be performed if it applies identically at every level of the recursion, and then it must be applied at all levels simultaneously.

Then $\boldsymbol{T}(S\boldsymbol{D})$ is finite, because all applications of a declaration inside a recursion generate together at most one transformation $\boldsymbol{T}'$ from any $\boldsymbol{T}$ for S. By definition, only recursion can make a transformation set infinite.

Weaker restrictions are conceivable. A declaration inside a recursion may generate any finite number of transformations to keep $\boldsymbol{T}(S\boldsymbol{D})$ finite. For a program whose fastest trace sets are **not** within a finite transformation set, see Sect. 7.1.

Here is a collection of commands for the semantic transformation of trace sets:

rule	command	semantics
(G1)	discard v	drop node v
(G2)	commute v left/right	swap node v with left/right neighbour
(G3i)	ravel v left/right	merge node v and its left/right neighbour in a concurrent command
(G3 ii)	ravel v right trace 1/2	append node v to the left of trace 1/2 of its right neighbour (which must be a concurrent command)
(G3 iii)	ravel v left trace 1/2	append node v to the right of trace 1/2 of its left neighbour (which must be a concurrent command)

Consider, for example, the transformation sequence which generates concurrent trace $\tilde{\tau}_n$ from trace τ_n of *sort* n (Sect. 5.4):[†]

$$\overset{n}{\underset{i=3}{\vdots}} [\overset{i-3}{\underset{j=0}{\vdots}} [\overset{i-3-j}{\underset{1}{\vdots}} \text{commute } cs\ i-j \text{ left ; ravel } cs\ i-j \text{ left }]]$$

where $cs\ i$ is the node representing the first instance of component $cs\ i$ to the right of the node currently looked at. Thus $\tilde{\tau}_n$ can be derived from τ_n in one left-to-right pass of right-to-left commutations and ravellings in time $O(n^3)$ ($O(n)$ for every nested **for**). However, a bounded number of transformations can, if we permit unbounded input, improve the execution time only by a constant factor, never by a degree of magnitude.

To make improvements of orders of magnitude, we have to be able to recognize recursive patterns in transformation sequences and apply them to the recursive representation of trace sets. The automatic recognition of these patterns is a formidable problem and beyond the scope of this thesis. Our current alternative is human intervention. The user should be able to communicate recursive patterns in trace set transformations to the compiler. One way of

[†] $\tilde{\tau}_i = \tau_i$ for $i=1,2$. Therefore semantic transformations start with $n=3$.

doing this is by proving lemmas like the one in Sect. 5.4 and feeding the compiler the resulting trace sets. We might be able to perform at least syntactic transformations automatically, e.g., discover the identities for τ'_{n+1} and τ''_{n+1} of the proof in Sect. 5.4. Another way is to feed the compiler a parameterized transformation sequence like the previous with parameter n for *sort n*. It would then derive the transformed trace set by applying this transformation, in our case to $\tilde{\lambda}_n$ (see Sect. 5.4), but without evaluating the recursion parameter.

Improvements of orders of magnitude for unbounded input are a strong requirement - too strong for any specific application: machines will always be limited in their storage and processor capacity. If we know the application will never deal with more than k inputs, say, we may prefer to simply execute a transformation sequence without worrying about recursion. For example, a reduction of the execution time of a frequently used *sort n* from $O(n^2)$ to $O(n)$, $n \leq k$, may well be worth expending time $O(k^3)$. But, knowing that there is a transformation sequence of $O(n^3)$, we must still find it. The easy way is again to let the programmer specify it, but then we may as well use the supplied knowledge of recursion. Without recourse to the programmer, a left-to-right pass of right-to-left commutations and ravellings works for *sort n*, and looks like a reasonable approach for other programs as well. But, without experimental data, nothing more can be said.

6.2.2 At Run Time

Ideally, we would like to resolve all semantic declarations statically, before run time. But the right choice of trace set may depend on factors which are difficult to predict, like input data or time properties. If a sufficient execution time is already guaranteed, some remaining decisions are better resolved during execution.

According to our experience in using this methodology for program development, many programs contain only global declarations. The most likely breach of globality is an enabling condition for conditional commutativity or independence. An enabling condition can be thought of and implemented as a case analysis of trace sets. If the condition is satisfied the according transformation takes effect; otherwise it does not. If both alternatives have been compiled, the only additional operations at run time are the test of the enabling condition and the following branch.

The only run-time technique for trace set selection that we investigate in detail is racing. Assume a set of operations that are independent up to a point determined by their progress relative to each other. Instead of having the compiler select a trace set based on predicted execution times, we can race the operations during program execution to their point of dependence.

In RL, such a situation involves independence declarations that mutually exclude each other. The mutual exclusion of semantic declarations is to be distinguished from the mutual exclusion of program components. A set of semantic declarations is ***mutually exclusive*** if the application of any one declaration in the set renders the other set members unexploitable. The following RL program offers a possibility to race components A and B:

```
        S: A ; B
        A: A1 ; A2
        B: B1 ; B2
(1)     A & B
(2)     A || B1
(3)     B || A1
```

Imagine, for instance, A and B sharing a variable in their second component, while working in distinct variables in their first.

Commutativity (1) makes both independence declarations (2) and (3) exploitable. But whenever one is applied the other cannot be. Note that (2) and (3) together can be refined to

(4) $A1 \parallel B1$

(5) $A1 \parallel B2$

(6) $A2 \parallel B1$

A ***race*** of A and B makes at run time a choice between the mutually exclusive declarations (5) and (6): start $A1$ and $B1$ in parallel, applying (4); if $B1$ terminates first continue in parallel with $B2$, applying (5); if $A1$ terminates first continue in parallel with $A2$, applying (6). In the described program, racing A and B always yields an optimal execution:

If $B1$ is faster than $A1$, trace $\tau 1 =_{df} \left\langle {}_{B1} \overset{A1}{\rightarrow} {}_{B2} \right\rangle \rightarrow A2$ is selected,

if $A1$ is faster than $B1$, trace $\tau 2 =_{df} \left\langle \underset{B1}{A1 \rightarrow A2} \right\rangle \rightarrow B2$ is selected.

Their execution times are:

$$\begin{aligned} \Upsilon(\tau 1) &= \max(\Upsilon(A1), \Upsilon(B1)+\Upsilon(B2)) + \Delta_{<>} + \Upsilon(A2) \\ &= \begin{cases} \Upsilon_1^{B1} =_{df} \Upsilon(A1)+\Delta_{<>}+\Upsilon(A2) & \text{if } \Upsilon(B1) \leq \Upsilon(A1) \\ \Upsilon_1^{A1} =_{df} \Upsilon(B1)+\Upsilon(B2)+\Delta_{<>}+\Upsilon(A2) & \text{if } \Upsilon(A1) \leq \Upsilon(B1) \end{cases} \end{aligned}$$

$$\begin{aligned} \Upsilon(\tau 2) &= \max(\Upsilon(A1)+\Upsilon(A2), \Upsilon(B1)) + \Delta_{<>} + \Upsilon(B2) \\ &= \begin{cases} \Upsilon_2^{A1} =_{df} \Upsilon(B1)+\Delta_{<>}+\Upsilon(B2) & \text{if } \Upsilon(A1) \leq \Upsilon(B1) \\ \Upsilon_2^{B1} =_{df} \Upsilon(A1)+\Upsilon(A2)+\Delta_{<>}+\Upsilon(B2) & \text{if } \Upsilon(B1) \leq \Upsilon(A1) \end{cases} \end{aligned}$$

Because $\Upsilon_1^{B1} \leq \Upsilon_2^{B1}$, $\tau 1$ is the better choice if $B1$ is faster than $A1$,
because $\Upsilon_2^{A1} \leq \Upsilon_1^{A1}$, $\tau 2$ is the better choice if $A1$ is faster than $B1$.

In more complicated situations, racing may not produce an optimal execution. In fact, it can be arbitrarily bad. Assume, for instance, that A and B regain their independence at some point:

S: A ; B

A: $A1$; $A2$; $A3$

B: $B1$; $B2$; $B3$

(1) $A \mathbin{\&} B$

(2) $A \parallel \{B1, B3\}$

(3) $B \parallel \{A1, A3\}$

If $B1$ is faster than $A1$, trace $\tau 1 =_{df} \left\langle \genfrac{}{}{0pt}{}{A1}{B1 \rightarrow B2} \right\rangle \rightarrow \left\langle \genfrac{}{}{0pt}{}{A2 \rightarrow A3}{B3} \right\rangle$ is chosen,

if $A1$ is faster than $B1$, trace $\tau 2 =_{df} \left\langle \genfrac{}{}{0pt}{}{A1 \rightarrow A2}{B1} \right\rangle \rightarrow \left\langle \genfrac{}{}{0pt}{}{A3}{B2 \rightarrow B3} \right\rangle$ is chosen.

If we take, for example, the following execution times for A and B,

$$\Upsilon(A1) = 2 \quad \Upsilon(A2) = 1 \quad \Upsilon(A3) = 20$$
$$\Upsilon(B1) = 1 \quad \Upsilon(B2) = 20 \quad \Upsilon(B3) = 1$$

sequential execution takes 45, trace $\tau 1$ 42, and trace $\tau 2$ 24 time units. A race would select $\tau 1$ because $B1$ is faster than $A1$, achieving almost none of the speed-up that is possible. Of course, the long operation $B2$ on shared data should be in parallel with the long independent tail component $A3$.

Racing does not work here because the execution time of only the initial independent parts of A and B are determining the selection. For components which gain or regain independence after a period of dependence, methods with look-ahead are required. But this may be too costly at run time.

6.3 Example: Sorting (Networks)

For the purpose of semantic transformations, we were able to interpret the trace set of refinement *sort n* (Sect. 2.3) as a single trace (Sect. 5.4). Knuth calls *sort n* a sorting network. A ***sorting network*** is a sorting refinement with a "homogeneous sequence of comparisons, in the sense that whenever we compare K_i versus K_j, the subsequent comparisons for the case $K_i<K_j$ are exactly the same as for the case $K_i>K_j$, but with i and j interchanged" [KnuIII]. The alternations in such refinements are hidden inside ***comparator modules*** (in *sort n* they are the components *cs i*) which, when considering concurrency, can be viewed as basic statements. For the purpose of semantic transformations, trace sets for sorting networks reduce to single traces.

Here is another refinement with the same semantics as *sort n*. We add semantic declarations for similar concurrency:

```
other sort n:  $\mathop{;}\limits_{i=1}^{n} S\,i$
S 0:           skip
S i:           if $a[i-1]>a[i]$ then swap i ; $S\,i-1$ fi
swap i:        $t[i]:=a[i-1]$; $a[i-1]:=a[i]$; $a[i]:=t[i]$
```

(1) $\bigwedge_{i,j} j \neq i-1,i,i+1$: *swap* $i \parallel a[j-1]>a[j]$

(2) $\bigwedge_{i,j} j \neq i-1,i,i+1$: *swap* $i \parallel$ *swap* j

other sort n has the same order of execution time as ***sort n***: $O(n^2)$ for the refinement, $O(n)$ for the given semantic version. The exact time properties of ***other sort n*** are better than those of ***sort n***: ***other sort n*** omits obsolete comparisons. But it is not a sorting network, and every trace must be transformed individually.

The declarations of both ***sort n*** and ***other sort n*** obey restrictions I and II.

6.4 Trace Machines

In this section we describe machines that can execute trace sets. The execution of a well-formed trace set differs from that of one of its traces only by some guard evaluations (see Sect. 6.1). Their implementation is not our concern. We are more interested in the implementation of concurrent commands. We therefore consider "trace" machines rather than "trace set" machines, building on the fundamental work of Conway [Con63]:

Independence relations are binary, so we only consider binary concurrency. Concurrent commands with more than two tines could be faster than the according nesting of binary concurrent commands, but we will not concern ourselves with this optimization. It is rather trivial on both the language and the machine level.

We keep track of the level of concurrency in a trace execution by way of a

concurrency tree. A tine τ appears in the tree at the time of its execution. When the concurrent command $<^{\tau 1}_{\tau 2}>$ of tine τ is executed, $\tau 1$ and $\tau 2$ appear as sons of τ. A ***tine counter*** identifies the concurrent command and records the progress of its execution. The counter is shared by all tines of the concurrent command.

The program trace can be viewed as an isolated tine and builds the root of the tree. It is linked to a tine counter that records the progress of the call. The concurrency tree expands as follows:

(1) At the time of program call, the program trace is established as root and linked to an external tine counter with value 1.

(2) On initiation of a concurrent command $<^{\tau 1}_{\tau 2}>$ in trace τ, $\tau 1$ and $\tau 2$ are appended to τ and linked to a shared tine counter with value 2.

(3) On termination of a tine, the tine counter linked to it is decremented. A zero tine counter indicates termination of the according concurrent command.

We can implement this mechanism in a straight-forward manner:

We assign one processor to each tine. To start its execution, we provide the processor with the tine's starting address and a link to a tine counter. A concurrent command is executed by setting up a tine counter and starting its execution of the tines on different processors. Decrementing a tine counter to 0 resumes the execution of the tine that contains the concurrent command.

This implementation can be canonically mapped on a configuration of distributed processors without shared memory and with message passing as communication device. Every variable is stored with the processor assigned to the tine in which the variable is introduced, with the assumption that this is the smallest program part containing all tines that share the variable. During the execution of the concurrent command, the processor that initiated it has to be available for communication with the units processing its tines , in case they access shared variables. (It may be fastest to pass copies of the shared variables for local update to the tines and let them report changes back at their termination.

The change v' of a concurrent command $< {\tau 1 \atop \tau 2} >$ on a global variable with value v passed to $\tau 1$ and $\tau 2$ can then be synthesized by calculating $v' = v1 + v2 - v$, where vi is the value passed back by τi).

The static assignment of processors to tines is fast and simple, but uses unnecessarily many processors: one for each node in the concurrency tree. The maximum number of parallel computations is the number of leaves in the tree. We can optimize the utilization of processors by reassigning processors between tines:

(1) At the time of program call, some free processor is selected to start executing the program trace.

(2) On encountering a concurrent command, a processor requests the selection of two free processors for execution of its tines and deactivates itself.

(3) At termination of a tine, the executing processor deactivates itself but, in case that the concurrent command is terminated, only after a request to let some free processor continue the execution of the tine containing the concurrent command.

A special-purpose processor, the dispatcher, handles requests for processor activation.

This implementation is not appropriate for use on a distributed configuration because that would imply swapping of a tine's environment at each processor reassignment. It can be better mapped on a centralized configuration with shared memory:

Assume a cache-like paging mechanism for the assignment of memory modules to processors. Potentially, each processor can own any module.† Store the trace such that independent tines occupy discrete modules. With each tine all its data and only its data are stored. A processor assigned to a tine may access the modules the tine occupies. Since it cannot look at global data, the copying scheme described previously for the distributed configuration has to be

† Conway describes such a mechanism [Con63].

adopted. Presumably, access of global data means shared access of a memory module. The described mapping abolishes shared access. Processors executing in parallel will be assigned distinct modules.

If we drop the copying scheme and allow a processor to look also at modules which contain data not local to its tine, shared accesses are a possibility, and prerequisites of the used non-interference relation have to be enforced. (We require indivisibility of memory reference.)

7 More Examples

At this point, all of the formalism has been introduced, but several aspects of the methodology need to be demonstrated.

We will present a number of popular examples on which methodologies and programming languages with concurrency are usually tested. Unfortunately, people are often more interested in the behaviour of such programs than in a result. They essentially want to simulate activities of concurrency, synchronization, and scheduling. Our methodology does not apply to simulation - we do not specify behaviour. But we can express the desired concurrency, as we informally understand it, in RL.[†]

The formal treatment of the examples is deferred to appendices. We encourage the reader to look at them; they also provide new clues.

7.1 The Sieve of Eratosthenes

The odd prime numbers less than a given integer, N, are to be determined by rectifying the initial assumption that all odd numbers are prime. We specify:

$$\begin{array}{ll} \textit{sieve}.\textbf{pre}: & \bigwedge_{i \in I} \textit{prime}[i] \\ \textit{sieve}.\textbf{post}: & \bigwedge_{i \in I} (\textit{prime}[i] \equiv i \text{ is prime}) \end{array}$$

$$\text{where } I =_{df} \{ i \mid 3 \le i < N,\ i \in \mathbf{N},\ i \text{ odd} \}$$

This is a second "pure" problem, i.e., a problem which is fully specified by an input/output assertion pair. Our program follows suggestions of Knuth [KnuII] (for a proof of correctness see App. A.2):

[†] Some of the behaviours are caused by space limitations which could be properly specified, much like execution time, but are at present not taken into account.

$$\textit{sieve}: \quad \mathop{\vdots}_{i=1}^{\lfloor\frac{\sqrt{N}-1}{2}\rfloor} \ \textbf{if}\ \textit{prime}[2i+1]\ \textbf{then}\ \textit{elim mults of}\ 2i+1\ \textbf{fi}$$

$$\textit{elim mults of } i: \quad \mathop{\vdots}_{j_i=0}^{\lfloor\frac{N-i^2}{2i}\rfloor} \ \textit{prime}[i^2+2ij_i]:=\textbf{false}$$

(1) $\bigwedge_{i,j} i\neq j$: $\quad$ $\textit{prime}[i]:=\textbf{false} \parallel \textit{prime}[j]$

(2) $\bigwedge_{i,j} i\neq j$: $\quad$ $\textit{prime}[i]:=\textbf{false} \parallel \textit{prime}[j]:=\textbf{false}$

(3) $\bigwedge_{i}$ $\quad$ $!\ \textit{prime}[i]:=\textbf{false}$

Any pair of tests and/or eliminations of different numbers is independent; any test or elimination of a number is idempotent. (Remember that guard independence and idempotence need not be declared.)

According to the semantic declarations, guard $\textit{prime}[j]$ and statement *elim mults of* i are independent unless j happens to be a multiple of i. Thus the order of execution of $\textit{prime}[j]$ and *elim mults of* i may only be manipulated if j is not a multiple of i: transformed computations must maintain an order, introduced by the refinement, in which guard $\textit{prime}[j]$ appears only where its truth ensures that j is prime. In other words, no computation will call *elim mults of* i for non-prime i. Moreover, because of idempotence, there are computations which eliminate every multiple exactly once.

The execution time of refinement *sieve* is exponential in the input N. The fastest computations for the present semantic version have an execution time constant in N: the time it takes to eliminate one multiple. Multiples are the index values of the second **for** loop. Their calculation is disregarded in the properties of *sieve* but is also of constant time, at the expense of exponential space.[†] (We agreed to initiate a **for** loop with the concurrent calculation of all its index values; see Sect. 4.2.1.)

[†] Other time/space relationships have to be coded explicitly in a recursive refinement. The *sieve* presented in [LeHe81] runs in $O(N)$ space and $O(N)$ time by allowing for concurrent elimination of multiples of different primes but requiring sequential elimination of multiples of the same prime.

One purpose of this example is to demonstrate how difficult optimum concurrency can be - even when derived from simple semantic declarations.

In this version of *sieve* there is no hope for optimum concurrency. To derive fastest computations for every input n, all multiples would have to be known - which makes the execution of *sieve* obsolete. Moreover, there is no way to find all multiples other than by infinite enumeration.

But we can compile *sieve* with limited concurrency. We know, for instance, that, if the multiples of the first m primes have been eliminated, the $m+1$st and $m+2$nd element left in I must be prime: the $m+1$st certainly is, the $m+2$nd must be because between a prime and its first multiple there is always another prime.† Therefore operations *elim mults of i* can, with increasing i, at least proceed pairwise in parallel. ‡

We leave it up to the number theorists to find more compilable concurrency. The trouble with prime numbers is that there are many properties that can only be proved with some margin of uncertainty. Such properties may be useful if taking a multiple for prime, once in a while, does not hurt. However, for this *sieve*, they are of no help. As pointed out previously, the refinement detects primes with certainty, and our calculus does not tolerate semantic deviations due to concurrency.

7.2 The Dining Philosophers

Five philosophers, sitting at a round table, alternate between eating and thinking. When a philosopher gets hungry, he picks up two forks next to his plate and starts eating. There are, however, only five forks on the table, one between each two philosophers. So a philosopher can only eat when neither of his neighbours is eating. When a philosopher has finished eating, he puts down his forks and goes back to thinking.

† Bertrand's Postulate [HaWr60]

‡ For a process program exploiting this fact see [Hoa75].

This example demonstrates how the methodology can be used to build never-ending algorithms. That is the only point we want to make in this section. A full formal treatment of the program modelling the five philosophers can be found in App. A.3.

Our methodology can only yield terminating refinements. Therefore we modify the problem specification and give the philosophers a finite life of N meals (allowing different philosophers individually many eating sessions is a trivial extension):

lives. **pre**:	all forks are on the table
lives. **post**:	all forks are on the table, and every philosopher has eaten exactly N times in this life

We represent the philosophers' actions by statements up_i and $down_i$ for a movement, i.e., seizure and release of fork i, eat_i for an eating session, and $think_i$ for a thinking period of philosopher i (for their refinements see App. A.3):[†]

$$lives: \quad \overset{N}{\underset{1}{;}} [\overset{4}{\underset{i=0}{;}} phil_i]$$

$$phil_i: \quad up_i ; up_{i\oplus 1} ; eat_i ; down_i ; down_{i\oplus 1} ; think_i$$

(1)	$\bigwedge_{i,j} j \neq i:$	$phil_i \ \& \ phil_j$
(2)	$\bigwedge_{i,j} j \neq i\ominus 1, i, i\oplus 1:$	$eat_i \parallel eat_j$
(3)	$\bigwedge_{i,j} j \neq i, i\oplus 1:$	$eat_i \parallel \{up, down\}_j$
(4)	$\bigwedge_{i,j} j \neq i:$	$\{up, down\}_i \parallel \{up, down\}_j$
(5)	$\bigwedge_{i,j} j \neq i:$	$think_i \parallel phil_j$

This program lets philosophers properly compete for their share of the meal and eventually die. The declarations state that philosophers may eat at different intervals according to their hunger (1), non-neighbours may eat at the same time (2), forks that are presently not used for eating may be moved (3),

[†] $\oplus$ denotes addition, $\ominus$ subtraction modulo 5.

different forks may be moved in parallel (4), and thinking philosophers do not interact with the rest of the system (5).

The total correctness of the refinement (see App. A.3) guarantees that the system cannot get stuck. None of the five semantic declarations invalidates total correctness (no correct declaration ever does), and therefore the concurrent version is also deadlock-free. We do not need additional proof, but to help the reader being convinced, here is a reasoning especially tailored for this algorithm: a situation where every philosopher has one fork and waits for the other cannot arise because in the refinement philosopher i lets no neighbour access the forks next to him once he prepares for eating, and none of the declarations permits commutations which would lift this restriction. The key is that (3) does not commute eating sessions with the movement of forks for neighbours.

For a never-ending program, a solution to the infinite problem, the finite *lives* may be called repeatedly in sequence. The user of our finite algorithms is responsible for a mechanism for infinite repetition; we refuse to consider it in our calculus. But we guarantee that, if the problem specification allows an infinite repetition of the solution (i.e., if the output assertion implies the input assertion), all algorithmic properties except termination are preserved. The user can rely on partial correctness, absence of deadlock, and absence of starvation without recourse to an authority outside the program, e.g., a fair scheduler.

Our solutions may be slightly more restrictive than non-terminating solutions have to be. We do not allow unbounded non-determinism, not even for unbounded activities. In this example, the table is cleared completely in arbitrarily long intervals, whereas it is not necessary for all philosophers to leave (or, in our terminology, die).

There is a straight-forward extension to our calculus which lifts this restriction: the exploitation of semantic relations between components of **different** calls of refinement *lives* must be allowed. Then a call of *lives* can start before the previous call terminates. Deadlock remains impossible, but to prevent starvation it must somehow be ensured that no call $phil_i$ is commuted to infinity.

This is the problem of "fair scheduling", which we shall not address.

7.3 Producing and Consuming

Suppose that two parts of a program are almost distinct; there is only one variable, buf, which appears in both. One part "produces" values and deposits them into the buffer, the other "consumes" deposited values by reading them from the buffer. The following program fragment reflects this situation. Productions and consumptions are called in turn M times; the omitted expressions in the assignments represent the creation of item i in production i, and the use of item i in consumption i:

$$stream: \overset{M-1}{\underset{i=0}{;}} [\, prod_i ; cons_j \,]$$

$$prod_i: \quad buf :=$$

$$cons_i: \quad := buf$$

The dependence of productions and consumptions forces an execution in sequence. But we would like some concurrency. We apply the independence theorem (Sect. 4.3.1) and disconnect different *prod/cons* pairs by giving everyone its private buffer. Then a production of some item i may go on in parallel with the consumption of other items j produced previously:

$$stream: \overset{M-1}{\underset{i=0}{;}} [\, prod_i ; cons_j \,]$$

$$prod_i: \quad buf[i] :=$$

$$cons_i: \quad := buf[i]$$

(1) $$\bigwedge_{i,j} i \neq j: \quad prod_i \parallel cons_j$$

We did not specify any special properties of productions and consumptions other than their interaction over buf. In cases where the creation of item i does not depend on knowledge of items 0 to $i-1$ productions may be mutually independent, and similarly may be consumptions. Then we may add two more declarations:

(2) $\bigwedge_{i,j} i \neq j:\ prod_i \,\|\, prod_j$

(3) $\bigwedge_{i,j} i \neq j:\ cons_i \,\|\, cons_j$

For an implementation we must assume a bound, say n, on the number of variables. We modify the previous program by indexing the buffer modulo n:

$$stream:\ \mathop{;}_{i=0}^{M-1} [\,prod_i ; cons_i\,]$$

$$prod_i:\quad buf[i \ \mathbf{mod}\, n] :=$$

$$cons_i:\quad := buf[i \ \mathbf{mod}\, n]$$

(1) $\bigwedge_{i,j} (i \neq j)\,\mathbf{mod}\, n:\ prod_i \,\|\, cons_j$

(2) $\bigwedge_{i,j} (i \neq j)\,\mathbf{mod}\, n:\ prod_i \,\|\, prod_j$

(3) $\bigwedge_{i,j} (i \neq j)\,\mathbf{mod}\, n:\ cons_i \,\|\, cons_j$

Concurrency works only for *prod/cons* pairs within a certain neighbourhood:[†] productions can be at most n items ahead of consumptions, then the buffer is full; consumptions can at most catch up with productions, then the buffer is empty. Note that the condition "buffer empty" is part of any program with producing and consuming, whereas "buffer full" arises out of a need for a buffer bound in an implementation.

The outcome of this section is nothing new. The solution to concurrent producing and consuming is well-known. But we wanted to exhibit its stepwise development in our methodology and tried not to rely on previous knowledge.

[†] More independence is declared but not exploitable.

8 Conclusions

8.1 Summary

We take the view that concurrency is a property not of programs but of executions. Consequently, looking at programs, parallelism may be hard to recognize. RL programs do not contain concurrent commands or synchronization primitives, only declarations of independence. And because an independence declaration may remain unexploited it only suggests that some action may or may not be involved in a parallel execution.

The reader might feel that starting with the definition of processes helps structuring a program, and that a refinement without immediate regard to concurrency forces us to artificially order logically separate tasks. We believe there is a separation of concerns: modularity structures the problem solution, concurrency speeds its execution up. For example, an airline reservation or taxi dispatching system receives its structure from the modularity of its transactions. It can well be imagined as an arbitrary sequence of transactions (in fact, this view will be helpful), although only a concurrent execution will be practical.

Our methodology yields modularity by way of refinement (as part of the language!) and concurrency by way of semantic declarations. A change in the refinement is likely to affect the semantic declarations for it (but an extension does not). Concurrency works bottom-up and is therefore susceptible to top-down design changes. But we insist the solutions are modifiable. They only put concurrency in its proper place: determined by, not determining the semantics of the refinement. If a refinement does not permit enough parallelism, the independence theorem advises to spread out variables over its components.

Concurrent actions are not synchronized by conditional delays but by conditional concurrency. The solutions are the same but our methodology prevents overdefinition and subsequent restriction of concurrency. The definition of concurrency proceeds step by step on semantically correct territory, successive declarations yielding faster and faster executions. Exclusion is not explicitly programmed. A process design approaches a solution from incorrect territory by trying to exclude wrong concurrency.

However, the concurrency of RL programs in its present form is less flexible than that of process programs. RL programs are not asynchronous as are process programs. Consider the process program

```
x:=0;
cobegin <x:=x+1>; S1
  //     S2 ; await < x>0 → "use x" >
coend
```

where $S1$ and $S2$ use distinct variables and do not use x. With execution times

$$\Upsilon(x{:=}0) = 1 \quad \Upsilon(x{:=}x+1) = 2 \quad \Upsilon(S1) = 4$$
$$\Upsilon(x>0) = 1 \quad \Upsilon(\text{"use } x\text{"}) = 2 \quad \Upsilon(S2) = 3$$

this program can be executed in at best seven time units. (Note that the condition $x>0$ is instantly satisfied when tested.) The corresponding RL program,

```
      S:  x:=0; A ; B
      A:  x:=x+1; S1
      B:  S2 ; "use x"
(1)       A || S2
(2)       B || S1
```

has three concurrent traces,

$$x{:=}0 \rightarrow x{:=}x+1 \rightarrow \left\langle {}^{\;S1}_{S2 \rightarrow \text{"use } x\text{"}} \right\rangle$$
$$x{:=}0 \rightarrow \left\langle {}^{x:=x+1}_{\;S2} \right\rangle \rightarrow \left\langle {}^{\;S1}_{\text{"use } x\text{"}} \right\rangle$$
$$x{:=}0 \rightarrow \left\langle {}^{x:=x+1 \rightarrow S1}_{\;S2} \right\rangle \rightarrow \text{"use } x\text{"}$$

whose executions all require eight time units.

The reason for this inoptimality is that we describe the semantics of RL programs by sequences of basic RL statements. A basic statement cannot be split further, e.g., between two concurrent commands as this example requires. To derive an optimal solution the refinement would have to be continued, maybe, to a very low machine level.

In our methodology proofs do not require auxiliary variables [OwGr76a, OwGr76b]. It seems auxiliary variables have the purpose of relating the histories of concurrent program parts that have been proved separately, like processes. In our programs concurrency is not synthesized from separate histories.

We consider only programming problems with results, i.e., we can only derive terminating programs. Infinite repetition is a mechanism of the user environment and applies only to entire RL programs, not their parts. Absence of deadlock is guaranteed by the total correctness of the RL program, and the question of starvation does not arise.

We do not permit program properties to vary due to overlapped execution. Consider the following program of two processes whose outcome depends on the point at which the first process is executed:

$$\textbf{cobegin}\ \ \langle x:=1\rangle\ //\ \textbf{do}\ \langle x=0 \rightarrow y:=y+1\rangle\ \textbf{od}\ \ \textbf{coend}$$

There is no equivalent in RL because different interleaved executions have different properties.[†]

An important application of this type is concurrent garbage collection as described for a LISP environment in [Gri77, Dij78]:

Assume a program part *mutation i* producing garbage, and a *collection* concurrently appending garbage to a list of free space. In a process program the concurrency of *mutation i* and *collection* can be proved as long as we expect afterwards only the garbage of mutations previous to i are collected. But in RL the corresponding declaration

[†] Note that limiting the number of repetitions,

$$\textbf{do}\ k\ \textbf{times}\ \langle x=0 \rightarrow y:=y+1\rangle\ \textbf{od}$$

or skipping the assignment to y,

$$\textbf{do}\ \langle x=0 \rightarrow \textbf{skip}\rangle\ \textbf{od}$$

still does not yield programs expressible in RL: the point of execution of $x:=1$ determines both the partial correctness and termination of the **do** loop.

$$\textit{mutation } i \parallel_R \textit{collection}$$

where $R \equiv_{df}$ garbage of mutations previous to i collected

is not legal. The interleavings of *mutation i* and *collection* do not have identical properties: in some *collection* will pick up the garbage produced by *mutation i*, in others it will not. *mutation i* cannot be proved commutative with *collection*'s search for more garbage. The process program works because there the outcome of guard evaluations may vary for different interleavings.

Programs whose properties vary for different interleavings are messy. Including them into RL would seriously complicate semantic relations and the properties of concurrent commands.

Semantic proofs are of linear complexity with respect to the length of the program. For every trace set, time proofs are linear. The complexity of a time proof for the program is determined by the effort expended on finding a satisfactory trace set, i.e., by the complexity of the search algorithm and transformation sequence used, or the comprehension of the user who performs the transformation on paper. If a satisfactory transformation can be found in linear time, the entire proof of the program is linear. Our examples are *sort n* and the Dining Philosophers.

8.2 Related Research

The introduction and summary relate our methodology to customary programming with processes. Throughout the thesis we refer to the proof methods for process programs by Owicki and Gries [OwGr76a, OwGr76b]. These methods are, however, not concerned with program development.

The view of programming with concurrency closest to ours is taken in [LaSi79]. Van Lamsweerde and Sintzoff use a concurrent command but begin with sequential semantics. Exclusions may be removed by way of correctness-preserving program transformations which are not described in detail. The concurrent processes are guarded commands (transitions). The guards (synchronizing conditions) make any order necessary for correctness explicit. Transitions are repeated forever, and non-trivial formal techniques are necessary to prevent deadlock and starvation.

Broy describes the semantics of concurrent programs by correctness-preserving transformations but elaborates only on the opposite direction: serializing concurrent statements [Broy80]. The transformations exploit a simple global independence relation.

Idempotence appears in [Heh80], but for programs with traditional concurrency features (concurrent commands and synchronization primitives). The implementation described there works only for last-action calls.

Path expressions [FlHa76], incorporated in the specification language for concurrent systems COSY [Lau79a] and recently extended to predicate path expressions [And79], reflect an approach inverse to ours: starting out with complete concurrency one declares relations, paths, that tighten sequencing. There exist several semantic definitions, one with transition networks, another with vector firing sequences somewhat like our traces [Lau79b]. But since no quality distinctions are made between different sequences, there is no selection problem. As a specification tool path expressions do not come with a proof system.

Jones recognizes the lack of development methods for programs with concurrency, he calls them "interfering programs", and proposes extensions to previous methodologies for sequential programs, what he calls "isolated programs" [Jon80]. The basic idea is to incorporate requirements for parallel correctness into the problem specification.

8.3 Further Research

One goal of this thesis was to deduce the concurrency in a program from its semantic properties. We express semantic properties in the weakest precondition calculus but are not quite satisfied with the definition of the most difficult property: non-interference.

It is evident that the relation $\nleftrightarrow$ is not "a simple convention" [OwGr76a]. Despite its syntactic looks it is a semantic condition, for example, when subscripted variables are involved. We would like to relate non-interference, as all other semantic relations, to a postcondition.

$\nleftrightarrow$ is of practical relevance, but $\overset{free}{\nleftrightarrow}$ is of theoretical interest. We know that $\nleftrightarrow$ is not the weakest condition for non-interference assuming memory interlock on variables. But can $\overset{free}{\nleftrightarrow}$ still be relaxed?

Our list of semantic relations is not exhaustive. For example, relaxing the equivalences in the weakest preconditions for commutativity to implications yields semi-commutativity [Hoa75]. Full semi-commutativity and semi-independence are defined accordingly. (Semi-idempotence does not seem very useful.) For a program with semi-relations semantics become an execution-dependent property. The refinement represents the computations with the strongest semantics. We know only $S\{R\} \supset \boldsymbol{T}\{R\}$ for every transformation $\boldsymbol{T}$.

We have discussed the Dining Philosophers and derived a solution with certain limitations. For instance, we assign each philosopher a fixed number of eating sessions in advance and do not allow unbounded non-determinism, not even for unbounded activities. Just how seriously our methodology constrains the specification and solution of problems like the Dining Philosophers or the Banker's Algorithm [Dij68, Lau79a] remains to be clarified. (We have developed a Banker's Algorithm, of sorts. It is not part of this thesis but may subsequently appear in a paper.)

Our programming calculus, as it stands, assumes a centralized machine architecture (a number of processors with shared memory). In Sect. 6.4 we point out that we can implement RL programs on distributed machines. The timing calculus can even account for speed differences of processors. But to model the time lags of inter-processor communication we need an additional rule for inter-processor assignments which has the semantics of language rule (L2) but different time properties. This would enable us to represent distributed computations. However, a methodology for programming distributed machines should make a behaviour-oriented approach and, at this point, our methodology does not. We tried to add behaviour specifications [LeHe81] but found no satisfactory formalism so far. Behaviours are much more complicated than semantic or time properties.

However, the most urgent work remains to be done on the implementation of the concept of semantic declarations. We take a first step towards formalizing a methodology that incorporates the derivation of concurrency rather than postulating the existence of some concurrency situation. This thesis provides a formal semantic model but is rather vague on its implementation. In our approach, the hard problem is the compilation, not the execution of the program. What algorithm can replace the intuition of the programmer in the search for a suitable trace set and still be of acceptable complexity?

8.4 On the Purpose of Programming Methodologies

One problem of formal semantics and verification is that even simple and seemingly obvious programs have lengthy and complicated proofs. Our programming methodology has been demonstrated on a set of simple "toy" problems. But the complexity of their formal treatment may give rise to the conclusion that we are unable to cope with larger, more realistic problems. The purpose of this section is to alleviate such scepticism.

Understanding the properties of a program to the last detail is a very strong requirement. A programming methodology is expected to adhere to it, and this adherence has to be demonstrated on examples which represent a class of common problems but are small enough to serve their tutorial purpose.

Verifying realistic applications to the last formal detail is a complex and tedious task which is always desirable but only feasible when the reliability of the solution is urgent enough to justify the expense.[†] Every program must be

[†] One would hope that automatic verification, if it can be made practical, will cut the expense drastically.

provable, but a full proof should not be demanded in every case.

Often only the crucial parts will receive a formal treatment; the rest will be derived with some degree of informality. And although there is no guarantee that informally derived program parts are totally reliable, a methodology should raise confidence that they will work sufficiently well. (The permitted margin of unreliability might determine the degree of informality in the program's derivation.)

We shall now argue that our methodology works for both formal and informal program development:

For every program, (actually, for every trace set), its precise properties are expressed by weakest preconditions with respect to any postcondition. Examples are the programs *fact n* (App. A.1) and, for semantic properties only, *sieve* (App. A.2) and *sort n* (Sect. 4.4).

If a weakest precondition is too hard to obtain, less will do as long as the problem specification is shown to be satisfied. Then the effects of the program are only known in environments described by this specification. An example is the derivation of only a worst-case execution time for program *sort n* (Sect. 4.4). For other than worst-case inputs, the exact execution time remains unknown.

An always complex weakest precondition is that of full commutativity: its derivation takes time proportional to the product of the lengths of the operands. We cope with that difficulty by applying the concept of globality and, in the simplest cases, employing the independence theorem. The majority of independence declarations will follow from the independence theorem and will not require a quadratic proof. We suggest: the user has to pay for complicated programs (here, for subtle concurrency).

One could be contented with a less formal idea about the properties of a solution or even have only an informal picture of the problem. Stepwise refinement is a fundamental and long-established method for careful informal program development. The same idea applied to the discovery of concurrency in a program yields stepwise semantic declarations. Each declaration can be understood in isolation and most declarations will be clear and simple. We presented two informal problems and their solutions (Sects. 7.2 and 7.3), and appended the formal work for one of them as a demonstration that our intuitions were correct (App. A.3). We believe that even a program with considerable proof length can in our methodology be simple on an informal basis.

9 References

[AFR80] Apt, K.R.; Francez, N.; de Roever, W.P.: " A Proof System for Communicating Sequential Processes", ***ACM TOPLAS*** 2, 3 (July 80), 359-385

[And79] Andler, S.: "Predicate Path Expressions", ***Proc. 6th Ann. Symp. on Principles of Programming Languages 79***, 226-236

[Broy80] Broy, M.: "Transformational Semantics for Concurrent Programs", ***Information Processing Letters*** 11, 2 (Oct 80), 87-91

[Con63] Conway, M.E.: "A Multiprocessor System Design", ***AFIPS Conf. Proc.*** 24, FJCC 63, 139-146

[Dij68] Dijkstra, E.W.: "Co-operating Sequential Processes", in ***Programming Languages***, F. Genuys (Ed.), Academic Press, 1968, 43-112

[Dij75] Dijkstra, E.W.: "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", ***Comm. ACM*** 18, 8 (Aug 75), 453-457

[Dij76] Dijkstra, E.W.: ***A Discipline of Programming***, Prentice-Hall, Series in Automatic Computation, 1976, 217 p.

[Dij78] Dijkstra, E.W. et al.: "On-the-Fly Garbage Collection: An Exercise in Cooperation", ***Comm. ACM*** 21, 11 (Nov 78), 966-975

[FlHa76] Flon, L.; Habermann, A.N.: "Towards the Construction of Verifiable Software Systems", ***Proc. 2nd int. Conf. on Software Engineering 76***, 141-148

[GCW79] Good, D.I.; Cohen, R.M.; Keeton-Williams, J.: "Principles of Proving Concurrent Programs in Gypsy", ***Proc. 6th Ann. Symp. on Principles of Programming Languages 79***, 45-52

[GeYe76] Gerhart, S.L.; Yelowitz, L.: "Observations of Fallibility in Applications of Modern Programming Methodologies", ***IEEE Trans. on Soft. Eng.*** SE-2, 3 (Sep 76), 195-207

[Gri77] Gries, D.: "An Exercise in Proving Parallel Programs Correct", ***Comm. ACM*** 20, 12 (Dec 77), 921-930, Corrigendum: ***Comm. ACM*** 21, 12 (Dec 78), 1048

[Gri78] Gries, D.: "The Multiple Assignment Statement", ***IEEE Trans. on Soft. Eng.*** SE-4, 2 (Mar 78), 89-93

[Gri81] Gries, D.: ***The Science of Programming***, Springer Verlag, Texts and Monographs in Computer Science, 1981, 366p.

[GrLe80] Gries, D.; Levin, G.: "Assignment and Procedure Proof Rules", ***ACM TOPLAS*** 2, 4 (Oct 80), 564-579

[HaWr60] Hardy, G.H.; Wright, E.M.: ***An Introduction to the Theory of Numbers***, 4th ed., Oxford University Press, 1960, p. 343

[Heh79] Hehner, E.C.R.: "**do** considered **od**: A Contribution to the Programming Calculus", ***Acta Informatica*** 11, 4 (1979), 287-304

[Heh80] Hehner, E.C.R.: "On the Design of Concurrent Programs", ***INFOR*** 18, 4 (Nov 80), 289-299

[Heh83] Hehner, E.C.R.: ***Programming Principles and Practice***, Prentice-Hall International, Series in Computer Science, to appear in 1983

[Hoa69] Hoare, C.A.R.: "An Axiomatic Basis for Computer Programming", ***Comm. ACM*** 12, 10 (Oct 69), 576-580, 583

[Hoa74] Hoare, C.A.R.: "Monitors: An Operating System Structuring Concept", ***Comm. ACM*** 17, 10 (Oct 74), 549-557, Corrigendum: ***Comm. ACM*** 18, 2 (Feb 75), 95

[Hoa75] Hoare, C.A.R.: "Parallel Programming: An Axiomatic Approach", ***Computer Languages*** 1, 2 (June 75), 151-160

[Hoa78a] Hoare, C.A.R.: "Some Properties of Predicate Transformers", ***Journ. ACM*** 25, 3 (July 78), 461-480

[Hoa78b] Hoare, C.A.R.: "Communicating Sequential Processes", ***Comm. ACM*** 21, 8 (Aug 78), 666-677

[Holt72] Holt, R.C.: "Some Deadlock Properties of Computer Systems", ***Computing Surveys*** 4, 3 (Sept 72), 179-196

[HoWi73] Hoare, C.A.R.; Wirth, N.: "An Axiomatic Definition of the Programming Language Pascal", ***Acta Informatica*** 2, 4 (1973), 335-355

[Jon80] Jones, C.B.: "Tentative Steps Towards a Development Method for Interfering Programs", Programming Research Group, Oxford University, Oct. 1980, 33p.

[KnuII] Knuth, D.E.: ***The Art of Computer Programming***, Vol. 2: ***Seminumerical Algorithms***, Addison-Wesley, 1969, p. 360, or: 2nd ed., 1981, p. 394

[KnuIII] Knuth, D.E.: ***The Art of Computer Programming***, Vol. 3: ***Searching and Sorting***, Addison-Wesley, 1973, p. 220 ff

[Kuck77] Kuck, D.J.: "A Survey of Parallel Machine Organization and Programming", ***Computing Surveys*** 9, 1 (Mar 77), 29-59

[Lam77] Lamport, L.: "Proving the Correctness of Multiprocess Programs", ***IEEE Trans. on Soft. Eng.*** SE-3, 2 (Mar 77), 125-143

[LaSi79] van Lamsweerde, A.; Sintzoff, M.: "Formal Derivation of Strongly Correct Concurrent Programs", ***Acta Informatica*** 12, 1 (1979), 1-31

[Lau79a] Lauer, P.E.; Torrigiani, P.R.; Shields M.W.: "COSY - A System Specification Language Based on Paths and Processes", ***Acta Informatica*** 12, 2 (1979), 109-158

[Lau79b] Lauer P.E.; Shields, M.W.; Best, E.: "Design and Analysis of Highly Parallel and Distributed Systems", in ***Abstract Software Specifications***, Lecture Notes in Computer Science 86, D. Bjørner (Ed.), Springer Verlag, 1979, 451-503, or: Tech. Rep. No. 142, Computing Lab., University of Newcastle-upon-Tyne, June 1979, 53 p.

[LeGr81] Levin, G.M.; Gries, D.: "A Proof Technique for Communicating Sequential Processes", ***Acta Informatica*** 15, 3 (1981), 281-302

[LeHe81] Lengauer, C.; Hehner, E.C.R.: "A Methodology for Programming with Concurrency", ***CONPAR 81***, Lecture Notes in Computer Science 111, W. Händler (Ed.), Springer Verlag, June 1981, 259-270

[Len78] Lengauer, C.: "On the Verification of Concurrent Algorithms", Tech. Rep. CSRG-94, Computer Systems Research Group, University of Toronto, Aug. 1978, 101 p.

[OwGr76a] Owicki, S.S.; Gries, D.: "An Axiomatic Proof Technique for Parallel Programs I", ***Acta Informatica*** 6, 4 (1976), 319-340

[OwGr76b] Owicki, S.S.; Gries, D.: "Verifying Properties of Parallel Programs: An Axiomatic Approach", ***Comm. ACM*** 19, 5 (May 76), 279-285

[Shaw79] Shaw, M.: "A Formal System for Specifying and Verifying Program Performance", Tech. Rep., Computer Science Dept., Carnegie-Mellon University, June 1979, 20 p.

[Sto67] Stone, H.S.: "One-Pass Compilation of Arithmetic Expressions for a Parallel Processor", ***Comm. ACM*** 10, 4 (Apr 67), 220-223

A Appendix: Formal Treatment of Programming Examples

A.1 The Factorial Program

We are interested in the properties of

```
fact n:  if $n=0 \to r:=1$
         [] $n>0 \to fact\ n-1;\ r:=r\cdot n$
         fi
```

where $\Upsilon(r\cdot n)=1$, $\Delta_{:=}=\Delta_{\text{if}}=\Delta_{\text{call}}=0$, and $\Upsilon(n)=\Upsilon(n=0)=\Upsilon(n>0)=\Upsilon(1)=\Upsilon(k)=0$, k being the actual index of the call.

Lemma:

$$fact\ k\ \{R\} \;\equiv\; k\geq 0 \wedge R^{r,\ clock}_{k!,\ clock-k}$$

Proof:

We need inductive approximations $(fact\ k)_i$.

Base: $(fact\ k)_0\{R\} \equiv$ **false**

For clarification, here is the first non-trivial approximation:

$$(fact\ k)_1\{R\} \equiv (\,n=0\wedge R^{r}_{1} \vee n>0\wedge \mathbf{false}\,)^{n}_{k}$$

$$\equiv k=0 \wedge R^{r}_{1}$$

ind. hyp.: $(fact\ k)_i\{R\} \equiv 0\leq k<i \wedge R^{r,\ clock}_{k!,\ clock-k}$

ind. step: $(fact\ k)_{i+1}\{R\} \equiv (\,n=0\wedge R^{r}_{1} \vee$

$$n>0\wedge (fact\ n-1)_i\{R^{r,\ \ clock}_{r\cdot n,\ clock-1}\}\,)^{n}_{k}$$

$$\equiv k=0\wedge R^{r}_{1} \vee k>0\wedge [\,0\leq k-1<i$$

$$\wedge\ (R^{r,\ \ clock}_{r\cdot n,\ clock-1})^{r,\ \ \ \ clock}_{(k-1)!,\ clock-(k-1)}\,]$$

$$\equiv 0\leq k<i+1 \wedge R^{r,\ clock}_{k!,\ clock-k}$$

$$\therefore \quad fact\ k\ \{R\} \;\equiv\; \bigvee_{i\geq 0} (fact\ k)_i\{R\} \;\equiv\; k\geq 0 \wedge R^{r,\ clock}_{k!,\ clock-k}$$

Corollary:

For positive k, call $fact\ k$ performs k multiplications.

Proof:

$$fact\ k\ \{clock \geq 0\} \;\equiv\; k\geq 0 \wedge (clock\geq 0)^{clock}_{clock-k} \;\equiv\; (k\geq 0,\ k\geq 0 \supset clock \geq k)$$

Thus, as function, $\mathrm{T}(fact\ k) = k$ with domain $k \geq 0$.

A.2 The Sieve of Eratosthenes

A proof with respect to specification

$$\begin{array}{ll} sieve.\,\mathbf{pre}: & \bigwedge_{i\in I} prime[i] \\ sieve.\,\mathbf{post}: & \bigwedge_{i\in I} (\,prime[i] \equiv i \text{ is prime}\,) \end{array}$$

$$\text{where } I =_{df} \{i \mid 3\leq i<N,\ i\in \mathbf{N},\ i \text{ odd}\}$$

is easy: by assignment rule (L2), the semantic weakest precondition of ***sieve*** is its postcondition with all occurrences of eliminated positions $prime[i]$ replaced by **false.** Thus

$$sieve\,\{sieve.\,\mathbf{post}\} \;\equiv$$

$$\bigwedge_{i\in I\setminus M} (\,prime[i] \equiv i \text{ is prime}\,) \wedge \bigwedge_{i\in M} (\,\mathbf{false} \equiv i \text{ is prime}\,)$$

$$\text{where } M =_{df} \{n \mid n\in I,\ \bigvee_{i\in I}\bigvee_{j\geq 0} n=i^2+2ij\}$$

The conjunct over M is true: M is the set of multiples in the odd numbers I. Therefore the weakest precondition reduces to

$$sieve\,\{sieve.\,\mathbf{post}\} \;\equiv\; \bigwedge_{i\in I\setminus M} prime[i]$$

such that $\quad sieve.\,\mathbf{pre} \supset sieve\,\{sieve.\,\mathbf{post}\}$.

A.3 The Dining Philosophers

If variable $eaten[i]$ counts the meals of philosopher i and variable $fork[i]$ indicates if fork i is currently on the table ($fork[i]=0$) or not ($fork[i]>0$), the semantic specification becomes:

lives. **pre**: $\bigwedge_{i=0}^{4} fork[i]=0$

lives. **post**: $\bigwedge_{i=0}^{4} (fork'[i]=0 \wedge eaten'[i]=eaten[i]+N)$

Resulting values of *fork* and *eaten* are primed, initial values are unprimed.

It should be clear that the following refinements of the philosophers' actions validate all five semantic declarations globally and complete the *lives* in conformity with the semantic specification:

up_i:	$fork[i]:=1$
$down_i$:	$fork[i]:=0$
eat_i:	$use\ fork\ i\ ;\ use\ fork\ i\oplus 1;\ eaten[i]:=eaten[i]+1$
$think_i$:	**skip**

$use\ fork\ i$ is an operation which touches variable $fork[i]$ but has no effect, for instance,

$use\ fork\ i$: **if** $fork[i]\geq 0 \rightarrow$ **skip fi**

The resulting RL program correctly simulates the philosophers' behaviour.

We presented these *lives* in Sect. 7.2 because they are attractive in their simplicity. But, although they exhibit the correct behaviour, their semantics are not adequate for the following reason:

There are more semantic relations which we did not declare because they generate undesirable behaviours. For example, philosophers do not have to alternate between eating and thinking; they may think at any time they wish. If we look at the declarations as program-specific with omitted enabling predicate $\bigwedge_k fork[k]\geq 0$, things are even worse: then, by dropping the qualifying predicate in (3), a philosopher may eat whether he has forks or not, and additional weakening of the qualifying predicate in (2) to $j \neq i$ lets neighbours eat at the same time.[†]

A model with adequate semantics should permit only desirable behaviours. We will strengthen the semantics of the philosophers' actions such that they may only be performed when the forks involved are in a proper state. Therefore different users of the same fork have to be distinguished.

Let variable $fork[i]$ indicate not only the position of fork i but also which philosopher j is holding it, if any ($fork[i]=j+1$). Let subscripts of operations *up, down, eat, think* denote strictly philosophers, and identify forks by superscripts relative to philosopher i (l for his left fork i, r for his right fork $i \oplus 1$):

```
lives:     $\overset{N}{\underset{1}{;}} [\, \overset{4}{\underset{i=0}{;}} phil_i \,]$
phil_i:    $up_i^l ; up_i^r ; eat_i ; down_i^l ; down_i^r ; think_i$
up_i^l:    if $fork[i]=0 \rightarrow fork[i]:=i+1$ fi
up_i^r:    if $fork[i\oplus 1]=0 \rightarrow fork[i\oplus 1]:=i+1$ fi
down_i^l:  if $fork[i]=i+1 \rightarrow fork[i]:=0$ fi
down_i^r:  if $fork[i\oplus 1]=i+1 \rightarrow fork[i\oplus 1]:=0$ fi
eat_i:     if $fork[i]=i+1 \wedge fork[i\oplus 1]=i+1 \rightarrow$
              use fork $i$ ; use fork $i\oplus 1$; $eaten[i]:=eaten[i]+1$ fi
think_i:   if $fork[i]\neq i+1 \wedge fork[i\oplus 1]\neq i+1 \rightarrow$ skip fi
```

(1) $\bigwedge_{i,j} j \neq i:$ $\quad phil_i \,\&\, phil_j$

(2) $\bigwedge_{i,j} j \neq i:$ $\quad eat_i \parallel eat_j$

[†] This illustrates that interpreting declarations as program-specific can affect their exploitability.

(3) (a) $\bigwedge_{i,j} j \neq i \ominus 1, i$: $\quad eat_i \parallel \{up, down\}_j^l$

(b) $\bigwedge_{i,j} j \neq i, i \oplus 1$: $\quad eat_i \parallel \{up, down\}_j^r$

(4) $\bigwedge_{j_1, j_2} \bigwedge_{k_1, k_2} \bigwedge_{i,m} \begin{bmatrix} k_1 & k_2 \\ j_1 & j_2 \end{bmatrix} \neq \begin{bmatrix} m & m \\ i & i \end{bmatrix}, \begin{bmatrix} r & l \\ i & i \oplus 1 \end{bmatrix}$:

$\{up, down\}_{j_1}^{k_1} \parallel \{up, down\}_{j_2}^{k_2}$

(5) $\bigwedge_{i,j} j \neq i$: $\quad think_i \parallel phil_j$

The proof of the refinement with respect to the semantic specification and of all declarations with respect to the global scope is left as an exercise to the reader. Note that simultaneous eating of neighbours can still be declared but not exploited because of a lack of forks (2).

University of Toronto
Computer Systems Research Group

BIBLIOGRAPHY OF CSRG TECHNICAL REPORTS 1980 - present

* - Out of print

* CSRG-108 DIALOGUE ORGANIZATION AND STRUCTURE FOR INTERACTIVE INFORMATION SYSTEMS
John Leonard Barron
[M.Sc. Thesis, DCS, 1980]

* CSRG-109 A UNIFYING MODEL OF PHYSICAL DATABASES
D.S. Batory, C.C. Gotlieb, April 1980

* CSRG-110 OPTIMAL FILE DESIGNS AND REORGANIZATION POINTS
D.S. Batory, April 1980

* CSRG-111 A PANACHE OF DBMS IDEAS III
D. Tsichritzis (ed.), April 1980

CSRG-112 TOPICS IN PSN - II: EXCEPTIONAL CONDITION HANDLING IN PSN; REPRESENTING PROGRAMS IN PSN; CONTENTS IN PSN
Yves Lesperance, Byran M. Kramer, Peter F. Schneider
April, 1980

CSRG-113 SYSTEM-ORIENTED MACRO-SCHEDULING
C.C. Gotlieb and A. Schonbach
May 1980

CSRG-114 A FRAMEWORK FOR VISUAL MOTION UNDERSTANDING
John Konstantine Tsotsos
[Ph.D. Thesis, DCS, June 1980]

CSRG-115 SPECIFICATION OF CONCURRENT EUCLID
James R. Cordy and Richard C. Holt
July 1980

CSRG-116 THE REPRESENTATION OF PROGRAMS IN THE PROCEDURAL SEMANTIC NETWORK FORMALISM
Bryan M. Kramer
[M.Sc. Thesis, DCS, 1980]

CSRG-117 CONTEXT-FREE GRAMMARS AND DERIVATION TREES AS PROGRAMMING TOOLS
Volker Linnemann
September 1980

CSRG-118 S/SL: SYNTAX/SEMANTIC LANGUAGE INTRODUCTION AND SPECIFICATION
R.C. Holt, J.R. Cordy, D.B. Wortman
CSRG, September 1980

CSRG-119 PT: A PASCAL SUBSET
Alan Rosselet
[M.Sc. Thesis, DCS, October 1980]

CSRG-120 PTED: A STANDARD PASCAL TEXT EDITOR BASED ON THE KERNIGHAN AND PLAUGER DESIGN
Ken Newman, DCS
October 1980

CSRG-121 TERMINAL CONTEXT GRAMMARS
Howard W. Trickey
[M.Sc. Thesis, EE, September 1980]

CSRG-122 THE APPROXIMATE SOLUTION OF LARGE QUEUEING NETWORK MODELS
John Zahorjan
[Ph.D. Thesis, DCS, August 1980]

CSRG-123 A FORMAL TREATMENT OF IMPERFECT INFORMATION IN DATABASE MANAGEMENT
Yannis Vassiliou
[Ph.D. Thesis, DCS, September 1980]

CSRG-124 AN ANALYTIC MODEL OF PHYSICAL DATABASES
Don S. Batory
[Ph.D. Thesis, DCS, January 1981]

CSRG-125 MACHINE-INDEPENDENT CODE GENERATION
Richard H. Kozlak
[M.Sc. Thesis, DCS, January 1981]

CSRG-126 COMPUTER MACRO-SCHEDULING FOR HIGH PRODUCTIVITY
Abraham Schonbach
[Ph.D. Thesis, DCS, March 1981]

CSRG-127 OMEGA ALPHA
D. Tsichritzis (ed.), March 1981

CSRG-128 DIALOGUE AND PROCESS DESIGN FOR INTERACTIVE INFORMATION SYSTEMS USING TAXIS
John Barron, April 1981

CSRG-129 DESIGN AND VERIFICATION OF INTERACTIVE INFORMATION SYSTEMS USING TAXIS
Harry K.T. Wong
[Ph.D. Thesis, DCS, to be submitted]

CSRG-130 DYNAMIC PROTECTION OF OBJECTS IN A COMPUTER UTILITY
Leslie H. Goldsmith, April, 1981

CSRG-131 INTEGRITY ANALYSIS: A METHODOLOGY FOR EDP AUDIT AND DATA QUALITY CONTROL
Maija Irene Svanks
[Ph.D. Thesis, DCS, February 1981]

CSRG-132 A PROTOTYPE KNOWLEDGE-BASED SYSTEM
FOR COMPUTER-ASSISTED MEDICAL DIAGNOSIS
Stephen A. Ho-Tai
[M.Sc.Thesis, DCS, January 1981]

CSRG-133 SPECIFICATION OF CONCURRENT EUCLID
James R. Cordy, Richard C. Holt
August 1981 (Version 1)

CSRG-134 ANOTHER LOOK AT COMMUNICATING PROCESSES
E.C.R. Hehner and C.A.R. Hoare, July, 1981

CSRG-135 ROBUST CONCURRENCY CONTROL IN DISTRIBUTED DATABASES
Derek L. Eager
[M.Sc. Thesis, DCS, October 1981]

CSRG-136 ESTIMATING SELECTIVITIES IN DATA BASES
Stavros Christodoulakis
[Ph.D. Thesis, DCS, December 1981]

CSRG-137 SATISFYING DATABASE STATES
Marc H. Graham
[Ph.D. Thesis, DCS, December 1981]

CSRG-138 IMPROVING THE PERFORMANCE OF DATA BASE SYSTEMS
Geovane Cayres Magalhaes
[Ph.D. Thesis, DCS, December 1981]

CSRG-139 A FORMAL TREATMENT OF INCOMPLETE KNOWLEDGE BASES
Hector J. Levesque
[Ph.D. Thesis, DCS, February 1982]

CSRG-140 AN OVERVIEW OF TUNIS: A UNIX LOOK-ALIKE
WRITTEN IN CONCURRENT EUCLID
R.C. Holt, February 1982

CSRG-141 ON PROVING THE ABSENCE OF EXECUTION ERRORS
W. David Elliott
[Ph.D. Thesis, DCS, September 1980]

CSRG-142 A METHODOLOGY FOR PROGRAMMING WITH CONCURRENCY
Christian Lengauer
[Ph.D. Thesis, DCS, April 1982]