Computer Science
University of Manchester

Enhancing the Tractability of Rely/Guarantee Specifications in the Development of Interfering Operations

Pierre Collette and Cliff B. Jones

Technical Report UMCS-95-10-3

1 2

Enhancing the Tractability of Rely/Guarantee Specifications in the Development of Interfering Operations

Pierre Collette and Cliff B. Jones

Department of Computer Science
University of Manchester
Oxford Road, Manchester, UK.
<{p.collette,c.b.jones}@cs.man.ac.uk>

Recent technical reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from `ftp.cs.man.ac.uk` in the directory `pub/TR`. The files are stored as PostScript, in compressed form, with the report number as filename. They can also be obtained on WWW via URL `http://www.cs.man.ac.uk/csonly/cstechrep/index.html`. Alternatively, all reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, UK.

†This work has been supported by funding from the UK EPSRC. We thank Ketil Stølen for his helpful comments on a preliminary draft of this report. We also thank Juan Bicarregui, Yih-Kuen Tsay, and especially Tim Clement for their careful reading of later versions.

Abstract

Various forms of assumption/commitment specifications have been used to specify and reason about the interference that comes from concurrent execution; in particular, consistent and complete proof rules relating to shared state operation specifications –with rely and guarantee conditions– have been published elsewhere. This report investigates methodological issues in the formulation of such specifications, and their way to record design decisions. This work aims at making the use of rely/guarantee conditions more tractable, both at the specification level and in the development towards code.

1 Introduction

Formal methods based on model-oriented specifications like VDM or B are applicable to the development of sequential operations. In such approaches, state components can be common to several operations but only one operation is executed at a time. A sequential operation can then be interpreted as a binary relation on the state space and specified with pre and post conditions; examples are given below but readers are assumed to be familiar with pre/post specifications in the style of VDM. In [Jon81], rely and guarantee conditions are proposed as an extension to cope with the specification and development of *concurrent* operations, a situation that occurs when operations sharing state components have overlapping executions. The necessary background about rely/guarantee specifications is recalled in this report – detailed expositions (including sound and complete proof systems) can be found in [Stø91]. The new insights here come from an emphasis on methodological issues. Theoretical aspects of rely/guarantee specifications are intentionally omitted in favour of suggestions that improve their practicability in the development of concurrent operations.

This research is concerned with imperative programs whose meaning can be discussed with respect to a set of states – say $s_i \in \Sigma$. The additional complexity of concurrent versus sequential operations is due to the presence of *interference*: operations access state components that can be modified by the execution of other operations during their own execution. This difference from sequential operations can be emphasized by looking at computations. A computation of a sequential operation can be viewed as a single transition

$$s_0 \xrightarrow{\pi} s_n$$

from a starting state s_0 to a final state s_n. Of course, there might be many intermediate states between s_0 and s_n but only the initial and final states can be accessed by other operations. The (superfluous) label π indicates that this transition is performed by the operation. In the presence of interference, a computation not only includes steps of the operation, but also steps from its environment (other operations). If the latter are labelled with ϵ, a computation can be represented by a sequence of transitions

$$s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \ldots$$

where each label l_i is either π or ϵ; computations that terminate have a finite number of π-labelled steps.[1]

Usually, termination in an acceptable state can only be ensured under assumptions about the initial state. In specifications of sequential operations, such *assumptions* are recorded in a pre condition, whereas the *commitments* of the operation (definition of acceptability) are recorded in a post condition. It is understood that the commitments are to be fulfilled when the assumptions are satisfied. Termination of an interfering operation in an acceptable state also requires assumptions about the initial state but this

[1] Whether these are finer or coarser grained steps is a key issue that is discussed further below; meanwhile the steps are referred to as the visible steps of an operation.

is not sufficient: one also needs assumptions about the interference from other operations (ϵ-labelled steps). Indeed, nothing reasonable can be expected from an operation whose environment modifies the state in an arbitrary way.

The use of assumption/commitment specifications in the development of concurrent systems is not restricted to the formalism discussed in this report: other examples are [AL93, BK85, Col94, JT95, KR93, MC81, PJ91, Sta86, ZdBdR84]. Some of the methodological issues raised in this report hopefully spread across examples and formalisms but the case study is only representative of one specific class of shared-state operations. In general, operations have both an *input/output* behaviour and a *reactive* behaviour. The former determines the result of an operation in terms of its inputs whereas the latter describes the way it interacts with other operations. This research focuses on operations for which the input/output behaviour is more important than the reactive behaviour; what really matters for these operations is their final result. Any classification is highly debatable but a possible characterisation is that, in the absence of interference, the same operations should be meaningful and specifiable with pre and post conditions. The case study illustrates this. Section 2 gives specifications for both the sequential and the interfering versions of the same operations; the latter are inevitably more sophisticated than the former but in both cases, what really matters is the input/output behaviour.

Section 2 illustrates the use of rely and guarantee conditions with top-level specifications from the case study; a brief sketch of the development is also given. No novelty appears in Section 2; in particular, the specifications are subject to substantial improvement –in accordance with the suggestions made– in the remainder of the report. Visible steps are defined in Section 3. Next, the use of data invariants and other useful invariant properties is advocated in Section 4. Finally, recommendations on writing specifications and on the refinement of operations towards code are proposed in Sections 5 and 6 respectively.

A warning. Sections 3 to 6 make constant use of excerpts from the case study sketched in Section 2. Although the report is intended to be self-contained, interested readers can find an account of all the development steps (from specifications to code) in the appendix. Proof obligations, hence proofs, are omitted.

2 Introduction to the Case Study

The problem of recording equivalence classes over a (finite) set T of elements occurs in a variety of contexts from controlling equivalent part numbers in manufacturing applications to tracking equivalence classes in cryptography. The two basic operations are TEST(a, b) that tests if a and b are elements of the same class and EQUATE(a, b) that merges the equivalence classes of a and b into a single class. An efficient implementation –both in terms of space and time– can be based on a representation which records equivalence classes as trees (one tree per class); there are various proposals for keeping the path lengths short within trees, and here a new operation CLEANUP(a) that shortens the path from a to its root in the tree is added.

The representation by trees and the introduction of the CLEANUP operation are clearly insights in the development; but this case study has been carried out without other insights than these. The deliberate ignorance of previous solutions and the hunger for interfering operations actually led to tackling a more general problem. The complexity of this problem increases with the degree of concurrency. In the absence of interference, algorithms for the operations can be designed quite easily, with the usual care for extreme cases in searching data structures. The task becomes more complex when CLEANUP interferes with EQUATE or TEST, by modifying the inner structure of trees. This development goes further in that it also permits the concurrent execution of TEST and EQUATE, and the concurrent execution of *several instances* thereof. During their execution, operations thus access trees which are reshaped and merged by others. Although the usefulness of such a high degree of concurrency is debatable, it provides a sufficiently complex problem to raise issues about the practicability of a development method.

In model-oriented developments, operations are first specified over an abstract state space, which is then progressively reified into more concrete ones until all operations are specified in terms of implementable data structures. This section gives specifications of TEST and EQUATE on abstract states. The unique state component is p: *T-partition* where

$$T\text{-}partition \;=\; \{p \in (T\text{-}\mathbf{set})\text{-}\mathbf{set} \mid is\text{-}partition(p)\}.$$

The type invariant $is\text{-}partition(p)$[2] indicates that the union of the sets in p is T and that sets are disjoint; each set in p records an equivalence class on T. VDM specifications for the sequential version of TEST and EQUATE are given first; rely and guarantee conditions are then introduced to cope with the concurrent execution of several instances of TEST with several instances of EQUATE.

2.1 Sequential Operations

Sequential operations can be specified by pre and post conditions; hooked variables in post conditions refer to the initial state. Those elements in the same class as a in p are denoted by $P\text{-}class(a, p)$; the predicate $P\text{-}equiv(a, b, p)$ stands for $P\text{-}class(a, p) = P\text{-}class(b, p)$. Specifications P-TEST$_0$ and P-EQUATE$_0$ should not require further explanation[3].

P-TEST$_0$ $(a\colon T, b\colon T)\; t\colon \mathbb{B}$
 rd p : *T-partition*
post $t \Leftrightarrow P\text{-}equiv(a, b, p)$

P-EQUATE$_0$ $(a\colon T, b\colon T)$
 wr p : *T-partition*
post $p = (\overleftarrow{p} \setminus \{P\text{-}class(a, \overleftarrow{p}), P\text{-}class(b, \overleftarrow{p})\}) \cup \{P\text{-}class(a, \overleftarrow{p}) \cup P\text{-}class(b, \overleftarrow{p})\}$

[2]Omitted definitions can be found in the appendix.

[3]Specifications are prefixed and subscripted; undecorated names are reserved for informal references to operations; collections of states and operations could be collected into VDM modules.

Note on data invariants. At this abstraction level, a state *is* a partition of T and thus there exists no 'state' in which two sets in p have a non-empty intersection. Nevertheless, this does not exclude an (inefficient) implementation of EQUATE that first copies all elements of one set into the other and then destroys the first set, thus creating intermediate sets with a non-empty intersection. The correctness of this implementation can be formally justified by a reification of the state space that removes the data invariant *is-partition*(p) and adds it as a conjunct to the pre and post conditions of EQUATE. A 'representation' state is then just a set of sets in T but those sets must form a partition when the operation terminates. In this sequential setting, data invariants can thus be considered as pre and post conditions.

2.2 Interfering Operations

Because of the (potential) concurrent execution of EQUATE, equivalence classes might be merged *during* the execution of TEST, and during the execution of another instance of EQUATE too. The high degree of interference is more apparent when equivalence classes are represented by trees (roots change, the inner structure of trees change), but interference can already be specified, hence better understood, at this abstraction level.

Consider the specification P-TEST$_1$ below, where the keyword **ext** indicates that p can be modified by other operations. Its *rely* condition asserts that classes only grow. This rely condition is thus an assumption about the interference of the environment (other operations) during the execution of TEST. It is interpreted as a reflexive and transitive binary relation that characterises any pair of states linked by an uninterrupted sequence of ϵ-labelled steps in a computation.

$$\textit{P-grows}(p_1, p_2) \quad \triangleq \quad \forall a: T \cdot \textit{P-class}(a, p_1) \subseteq \textit{P-class}(a, p_2)$$

P-TEST$_1$ $(a: T, b: T)\ t: \mathbb{B}$
 ext **rd** p : *T-partition*
rely $\textit{P-grows}(\overleftarrow{p}, p)$
post $(\textit{P-equiv}(a, b, \overleftarrow{p}) \;\Rightarrow\; t) \wedge (t \;\Rightarrow\; \textit{P-equiv}(a, b, p))$

An operation is only required to terminate and satisfy the post condition if the pre condition holds initially *and* the rely condition holds for all ϵ-labelled steps. Interference is thus explicitly specified but, as for the specification P-TEST$_0$ (without interference), the important part is the input/output behaviour. The post condition now consists of

- a sufficient condition that forces t to be **true** if a and b were members of the same class when the operation started; and
- a necessary condition that allows t to be **true** only if a and b are members of the same class when the operation terminates.

The result of the operation cannot be determined in the case where the classes of a and b are initially disjoint and then merged by a concurrent execution of EQUATE. Wrong post conditions can often be detected by the mandatory proof obligation that requires them to be preserved by interference. In this case, the proof obligation is

$$\frac{\begin{array}{c} p_0, p_1, p_2\colon T\text{-}partition \\ (P\text{-}equiv(a, b, p_0) \;\Rightarrow\; t) \wedge (t \;\Rightarrow\; P\text{-}equiv(a, b, p_1)) \\ P\text{-}grows(p_1, p_2) \end{array}}{(P\text{-}equiv(a, b, p_0) \;\Rightarrow\; t) \wedge (t \;\Rightarrow\; P\text{-}equiv(a, b, p_2))}$$

where the subscripts 0, 1, and 2 refer to the initial state, a possible final state, and another final state after additional interference from other operations. Note that, in the presence of arbitrary interference (i.e. no rely condition), the result would be absolutely unpredictable. In contrast, if no interference is allowed (i.e. a rely condition of $p = \overleftarrow{p}$), the post condition of P-TEST$_1$ reduces to the post condition of P-TEST$_0$.

The counterpart of the rely condition is the *guarantee* condition which specifies the interference to others caused by an operation; this is what other operations may rely upon. This guarantee condition is interpreted as a reflexive binary relation that holds for all π-labelled steps in a computation. Together with the post condition, it forms the commitments of the operation to its environment. There is no explicit guarantee condition in P-TEST$_1$ above but the mode restriction **rd** p guarantees that no step of the TEST operation modifies p. A guarantee condition appears in P-EQUATE$_1$: it asserts that classes only grow and no other classes than those of a and b can be modified by steps of the operation.

P-EQUATE$_1$ $(a\colon T, b\colon T)$
 ext **wr** p : $T\text{-}partition$
rely $P\text{-}grows(\overleftarrow{p}, p)$
guar $P\text{-}grows(\overleftarrow{p}, p) \wedge$
  let $rest = T \setminus (P\text{-}class(a, \overleftarrow{p}) \cup P\text{-}class(b, \overleftarrow{p}))$ **in**
  $\forall e \in rest \cdot P\text{-}class(e, p) = P\text{-}class(e, \overleftarrow{p})$
post $P\text{-}equiv(a, b, p)$

Here again, this specification can be shown –in the absence of interference– to specialise to the non-interfering case (P-EQUATE$_0$): the argument relies on the guarantee condition as well as the post condition.

Coexistence. Whenever two operations are intended to be executed in parallel, there is a coexistence proof obligation on their specifications by which it is verified that the interference caused by one operation is allowed by the other. In this case, several instances of TEST can be executed in parallel with several instances of EQUATE because TEST is a read-only operation and the guarantee condition of P-EQUATE$_1$ implies the rely conditions of P-EQUATE$_1$ and P-TEST$_1$.

2.3 Data Reification

The recording of equivalence classes in a representation based on trees is captured by the reification of partitions into forests. The concrete state contains a single component f: *T-forest* that maps[4] each (non-root) element to its father in the tree. The type invariant $is\text{-}forest(f)$ prevents f from containing cycles.

$$T\text{-}forest \quad = \quad \{f \in T \xrightarrow{m} T \mid is\text{-}forest(f)\}$$

Each tree in f represents an equivalence class. Those elements in the same tree as a are denoted by $F\text{-}class(a, f)$; $F\text{-}equiv(a, b, f)$ stands for $F\text{-}class(a, f) = F\text{-}class(b, f)$; $is\text{-}root(a, f)$ indicates that a is a root in f; $ancestors(a, f)$ is the set of elements on the path from a to its root (a not included).

Partitions can be easily retrieved from forests:

$$p \quad = \quad \{F\text{-}class(a, f) \mid a \in T \wedge is\text{-}root(a, f)\}.$$

In a second stage of reification, the forest is implemented by an array m from T to T; $m(a) = a$ indicates that a is a root; the set of roots in m is $rts(m)$. This step merely consists of technical manipulations that transform the type invariant *is-forest* into a state invariant on directly implementable data structures (arrays). The new data invariant is then

$$m\text{-}is\text{-}forest(m) \quad \triangleq \quad is\text{-}forest(rts(m) \mathbin{⩤} m)$$

and the forest retrieved from m is $fr(m)$.

$$fr : T\text{-}array \to T\text{-}forest$$

$$fr(m) \quad \triangleq \quad rts(m) \mathbin{⩤} m$$

$$\textbf{pre } m\text{-}is\text{-}forest(m)$$

2.4 Operation Refinement

First, the specifications of TEST and EQUATE over partitions are refined into operations over forests and a new operation CLEANUP is added. Next, operation refinements are carried out at the forest representation level. The resulting elementary operations (e.g. move up in the tree, connect two elements) are then translated into operations on the array m, which in turn are refined into code.

A typical operation refinement, which is often referred to in the sequel, is shown in Figure 1. Equivalence classes of a and b can be merged in a sequential EQUATE operation by computing their roots and connecting them. But this simple algorithm must be revised to cope with interference from concurrent EQUATE operations that might turn roots into inner elements of new trees. In this development of EQUATE, interference is allowed during the computation of roots but is precluded by a **protect** mechanism during the

[4] The VDM notation $m: A \xrightarrow{m} B$ indicates a finite map m with domain type A and range type B; $⩤$ is the operator for domain subtraction ($\textbf{dom}\, s \mathbin{⩤} m = \textbf{dom}\, m \setminus s$).

local $x, y\colon T; t\colon \mathbb{B}$ **in**
$x, y := a, b$
repeat
F-ROOT$_1(x)$‖F-ROOT$_1(y)$;
protect f **in** F-TEST-AND-CONNECT$_1(x, y, t)$
until t
end

Figure 1: Decomposition of F-EQUATE$_1$

TEST-AND-CONNECT operation. This operation checks (with result in t) if two elements are roots *at the same time*; in case $t =$ **true** the operation connects one element to the other and the loop in Figure 1 terminates. Detailed specifications of these operations are purposely postponed to Section 5 but impatient readers can always consult the appendix for an account of all specifications and development steps.

3 Visible Steps

The question of granularity arises as soon as interference is discussed; a detailed introduction to this problem with examples can be found in [MP92]. In this context, the question amounts to what are the π-labelled steps in a computation. This directly affects the interpretation of the guarantee condition of an operation (and of course the rely conditions of others). As discussed in next section, this also affects the interpretation of invariants.

A visible step of an operation is one that produces values relevant to other operations. These include the initial and the final values of its shared (non-local) variables (relevant for sequential composition) but must also encompass every *public* intermediate value of its shared variables. Each occurrence of a variable in the code of an operation can indeed be classified as either *public* or *private* [MP92]. An occurrence of a shared variable is private if the variable cannot be accessed by a concurrently executed operation, e.g. when it appears inside mutually exclusive code sections.

In this case study, all occurrences of the array m in the code are public and thus each assignment to m is a visible step. This however does not mean that all assignment statements are executed atomically. For example, a crucial assignment statement for detecting the termination of the computation of roots is $r := (m(z) = z)$. This statement has been safely introduced in the development with the assumption that other operations may interfere (and modify m hence the truth value of $m(z) = z$) between the read and write memory accesses; r and z are local variables. Imposing atomicity for all assignment statements would require a lot of synchronisation overhead to implement them (see e.g. [And91]). Such overheads should only be incurred when required and specified by the designer (e.g. using atomic brackets). The evaluation of expressions is not assumed to be atomic either: the expression $m(x) = x \wedge m(y) = y$ in the actual code of TEST is not supposed to be executed atomically but enough synchronisation has been introduced during the design to ensure that other operations may read but not modify m when this statement is executed.

4 Invariants for Interfering Operations

A development method is helpful only if it helps master the inherent complexity of a problem. The use of rely and guarantee conditions favours local reasoning but the first draft versions (not shown in this report) of the specifications of TEST and EQUATE were still too complex, hence their subsequent modification. Essentially, their gratuitous complexity was due to the lack of invariant properties. This section first discusses the application of data invariants (in the style of VDM) to interfering operations; it then introduces a new kind of invariant property.

4.1 Data Invariants

Without doubt, it is much easier to view *is-forest* as a data invariant rather than to repeat this predicate in almost every condition of each specification and in the pre condition of auxiliary functions (*is-root*, *ancestors*, ...). Explicit data invariants also bring insight to the problem. Data invariants are helpful in the development of sequential operations[Jon90] and remain so in the development of interfering operations.

In the specification of sequential operations, data invariants can be considered as implicit pre and post conditions on all operations on the state space. Since the initial and final values are the only visible values of a sequential operation, this means that data invariants are required to hold for all visible values of an operation. Remarkably, this is conceptually the same for interfering operations. Data invariants are still required to hold for all visible values; there are just more visible values than the initial and final ones. Preservation of an invariant by visible steps can thus be considered as an implicit guarantee condition on all operations (hence a rely condition as well). In this case study, the preservation of the invariant is ultimately verified for the assignments to m, one in EQUATE, and one in CLEANUP.

4.2 Evolution Invariant

Although helpful, data invariants are not enough. The complexity of a development can be further reduced by the use of another invariant property. It should be clear from the examples in Section 2 that the relation $P\text{-}grows(p, p')$ holds for *any* pair of states where p' follows p in a computation (no matter whether the intermediate steps are operation or environment steps). This relation between computation states can be recorded by an *evolution* invariant, that should appear just next to the data invariant, in the data part of a specification.

$$ev\text{-}T\text{-}partition(p_1, p_2) \;\triangleq\; P\text{-}grows(p_1, p_2)$$

This evolution invariant can be interpreted as a reflexive and transitive binary relation that characterises *any* pair of states in a computation of *any* operation, no matter whether the steps that separate them are π or ϵ-labelled. The evolution invariant can be viewed as an implicit guarantee condition on all operations, and thus an implicit rely condition as well. As illustrated by P-EQUATE$_2$, the evolution invariant does not have to be repeated in the individual specifications.

P-EQUATE$_2$ $(a\colon T, b\colon T)$
ext **wr** p : $T\text{-}partition$
guar **let** $rest = T \setminus (P\text{-}class(a, \overleftarrow{p}) \cup P\text{-}class(b, \overleftarrow{p}))$ **in**
$\quad\forall e \in rest \cdot P\text{-}class(e, p) = P\text{-}class(e, \overleftarrow{p})$
post $P\text{-}equiv(a, b, p)$

At the forest representation level, the evolution invariant records the fact that not only do classes grow but also trees can only shrink. No descendant of an element later becomes one of its ancestors, but new ancestors may still appear due to the possible merging of trees. The proof obligation that the evolution invariant on forests implies the one on partitions is easily discharged.

$$ev\text{-}T\text{-}forest(f_1, f_2) \quad \triangleq \quad F\text{-}grows(f_1, f_2) \wedge F\text{-}shrink(f_1, f_2)$$

where

$$\begin{aligned} F\text{-}grows(f_1, f_2) \quad &\triangleq \quad \forall a\colon T \cdot F\text{-}class(a, f_1) \subseteq F\text{-}class(a, f_2) \\ F\text{-}shrink(f_1, f_2) \quad &\triangleq \quad \forall a\colon T \cdot ancestors(a, f_2) \cap F\text{-}class(a, f_1) \subseteq ancestors(a, f_1) \end{aligned}$$

The evolution invariant at the array representation level just mimics the previous one:

$$ev\text{-}T\text{-}array(m_1, m_2) \quad \triangleq \quad ev\text{-}T\text{-}forest(fr(m_1), fr(m_2)).$$

With computations restricted by $ev\text{-}T\text{-}forest$, the three operations on forests are specified below. The guarantee condition in F-EQUATE$_1$ ensures that equivalence classes are merged only by connecting the root of a tree to another tree. The rely condition of F-CLEANUP$_1$ is not an evolution invariant because some steps of CLEANUP are obviously intended to modify the inner structure of trees. The same applies to its guarantee condition (equivalence classes are untouched and nothing but $f(a)$ changes) because f can be modified by the environment in other ways.

$$\begin{aligned} bodyunch(f_1, f_2) \quad &\triangleq \quad \forall a\colon T \cdot \neg is\text{-}root(a, f_1) \;\Rightarrow\; \neg is\text{-}root(a, f_2) \wedge f_2(a) = f_1(a) \\ rootunch(f_1, f_2) \quad &\triangleq \quad \forall a\colon T \cdot is\text{-}root(a, f_2) \;\Leftrightarrow\; is\text{-}root(a, f_1) \end{aligned}$$

F-TEST$_1$ $(a\colon T, b\colon T)$ $t\colon \mathbb{B}$
ext **rd** f : $T\text{-}forest$
post $(F\text{-}equiv(a, b, \overleftarrow{f}) \;\Rightarrow\; t) \wedge (t \;\Rightarrow\; F\text{-}equiv(a, b, f))$

F-EQUATE$_1$ $(a\colon T, b\colon T)$
ext **wr** f : $T\text{-}forest$

guar $bodyunch(\overleftarrow{f}, f) \wedge$
let $rest = T \setminus (F\text{-}class(a, \overleftarrow{f}) \cup F\text{-}class(b, \overleftarrow{f}))$ **in**
$\forall e \in rest \cdot F\text{-}class(e, f) = F\text{-}class(e, \overleftarrow{f})$

post $F\text{-}equiv(a, b, f)$

F-CLEANUP$_1$ $(a\colon T)$
ext **wr** f : $T\text{-}forest$

rely $bodyunch(\overleftarrow{f}, f)$

guar $rootunch(\overleftarrow{f}, f) \wedge \{a\} \lhd f = \{a\} \lhd \overleftarrow{f}$

post $\neg is\text{-}root(a, \overleftarrow{f}) \wedge \neg is\text{-}root(\overleftarrow{f}(a), \overleftarrow{f}) \;\Rightarrow\; f(a) \neq \overleftarrow{f}(a)$

The mandatory proof obligation that requires post conditions to be preserved by interference again increases confidence in the specifications. Its formulation for F-CLEANUP$_1$ reveals the role of the evolution invariant as both a post condition and a rely condition:

$$\frac{\begin{array}{c} f_0, f_1, f_2\colon T\text{-}forest \\ ev\text{-}T\text{-}forest(f_0, f_1),\, ev\text{-}T\text{-}forest(f_1, f_2) \\ \neg is\text{-}root(a, f_0) \wedge \neg is\text{-}root(f_0(a), f_0) \;\Rightarrow\; f_1(a) \neq f_0(a) \\ bodyunch(f_1, f_2) \end{array}}{\neg is\text{-}root(a, f_0) \wedge \neg is\text{-}root(f_0(a), f_0) \;\Rightarrow\; f_2(a) \neq f_0(a)}$$

Evolution invariants are not a novelty *per se*. Predicates that appear in the rely and guarantee conditions of all operations were already emphasised in [Stø91] (called there binary invariants). As explained in Section 5, there are advantages in moving them from the specifications of individual operations into the specification of the shared state. In fact, even the idea that properties of all computations can be attached to the definition of a state is not new. The state specification modules of [Mid93] include a dynamic constraint which is a temporal formula. Interestingly –in the detailed case study of [Mid93]– the temporal formula has precisely the form of an evolution invariant.

5 Writing Specifications

Based on lessons learned from the case study, this section presents a few guidelines on writing specifications. To understand their impact, there is some incentive to present a 'bad' specification of ROOT first. This operation is introduced in the development of both TEST and EQUATE.

BAD-ROOT$_0$

f : T-*forest*

z : T

rely $z = \overleftarrow{z} \wedge (connect(\overleftarrow{f}, f) \vee shorten(\overleftarrow{f}, f))$

guar $f = \overleftarrow{f} \wedge z = f(\overleftarrow{z})$

post $F\text{-}equiv(\overleftarrow{z}, z, f) \wedge is\text{-}root(z, \overleftarrow{f})$

The predicates *connect* and *shorten* in the rely condition capture the interference from EQUATE and CLEANUP respectively.

$$connect(f_1, f_2) \;\triangleq\; \forall a: T \cdot f_2(a) \neq f_1(a) \;\Rightarrow\; is\text{-}root(a, f_1) \wedge \neg F\text{-}equiv(a, f_2(a), f_1)$$
$$shorten(f_1, f_2) \;\triangleq\; \forall a: T \cdot f_2(a) \neq f_1(a) \;\Rightarrow\; depth(a, f_2) < depth(a, f_1)$$

The first drawback of BAD-ROOT$_0$ lies in its rely condition which focuses on other operations rather than on the assumptions needed by ROOT. The second drawback is the occurrence of z in the rely and guarantee conditions. No other operation is accessing z during the execution of ROOT and there should be no need to say anything about interference on z. Mentioning z in the guarantee condition has another nasty impact on subsequent developments: BAD-ROOT$_0$ can only implemented by an operation that computes the new value of z (going up the tree) in an atomic step. A careful analysis of the actual code developed from a better specification reveals harmless situations where, because of interference, the new value of z is not an ancestor of its old value. This unnecessary constraint on granularity is thus a third drawback of BAD-ROOT$_0$. Its fourth drawback is the lack of a clear indication that ROOT is a read-only operation on f; there should be no guarantee condition at all. In fact, the actual specification of ROOT has no rely condition either: all needed assumptions on interference are already captured by the evolution invariant. Although a bad specification, BAD-ROOT$_0$ is not the worst one: data invariants could be neglected and f be just a map from T to T, instead of a forest.

Moving away from this bad example, the rest of this section investigates methodological issues in writing rely-guarantee specifications. New examples are given but of course all specifications from Section 4 should be considered as 'good' examples too.

5.1 Usefulness of the Invariants

Data and evolution invariants do not increase the expressive power of specifications because they can be otherwise incorporated into specifications. Indeed, the data invariant holds initially (pre condition), is preserved by visible steps (rely and guarantee condition) and thus holds upon termination (post condition); the evolution invariant holds for every pair of visible steps (rely and guarantee condition) and by transitivity holds upon termination (post condition).

Yet, data and evolution invariants are not just syntactic sugar: they each bring insight into the problem. Having those invariants in mind helps the process of writing specifications. Interesting properties can also be deduced from the invariants. Typical examples are the irreversibility of the transformation of forests and the impossibility for new roots to be created:

$$\frac{\begin{array}{c} f_1, f_2, f_3\colon T\text{-}forest \\ ev\text{-}T\text{-}forest(f_1, f_2),\, ev\text{-}T\text{-}forest(f_2, f_3) \\ f_3 = f_1 \end{array}}{f_2 = f_1} \qquad \frac{\begin{array}{c} a\colon T, f_1, f_2\colon T\text{-}forest \\ ev\text{-}T\text{-}forest(f_1, f_2) \\ is\text{-}root(a, f_2) \end{array}}{is\text{-}root(a, f_1)}$$

The many roles of the evolution invariant (rely, guar, and post conditions) are especially useful in proofs. The first premise of most proofs is usually a list of state components, e.g. $f_0, f_1, f_2\colon T\text{-}forest$. The states in consideration can be the initial state, the intermediate state in a sequential composition, the states before and after a visible step of the operation (proof obligations for guarantee condition), a potential final state and a new one due to interference, etc. In all cases, they represent successive states in a computation and this means that

$$ev\text{-}T\text{-}forest(f_0, f_1),\, ev\text{-}T\text{-}forest(f_1, f_2),\, ev\text{-}T\text{-}forest(f_0, f_2)$$

can be freely used anywhere in the proof, just as

$$is\text{-}forest(f_0),\, is\text{-}forest(f_1),\, is\text{-}forest(f_2)$$

can be. Automatic inheritance of those predicates is convenient in proofs. A typical proof step is the verification of $F\text{-}equiv(a, b, f_2)$ from $F\text{-}equiv(a, b, f_1)$ knowing that f_2 occurs after f_1, e.g. when a suboperation and environment steps occur in between. This proof follows easily from $F\text{-}grows(f_1, f_2)$. The predicate $F\text{-}shrink(f_1, f_2)$ is typically used to prove $(is\text{-}root(a, f_2) \Rightarrow t)$ from $(is\text{-}root(a, f_1) \Rightarrow t)$ when f_2 follows f_1. Without explicit invariants, those predicates would have had to have been reconstructed separately from the guarantee conditions of the suboperations and from the overall rely condition.

In conclusion, although data and evolution invariants could be incorporated in the individual specifications of the operations, what eases the development process is precisely *avoiding* thinking about them in terms of assumptions and commitments. Invariants should be considered as *given* and available for free use in writing and reasoning about specifications. The same philosophy is adopted in [MV90]: the use of invariants in the design should be separate from their ultimate verification. How the latter is carried out is addressed in Section 6.

5.2 Enriched Mode Restrictions

Write-mode restrictions on variables can be understood as commitments of the operation: no other variables can be modified. Read-mode can be interpreted in several ways [Bic92]; in this case study, all variables that can be accessed but not modified by the operation are required to appear with read-mode; non-mentioned variables cannot be accessed by the operation. The mode restrictions also play a syntactic role: only the variables in write-mode can be hooked in post conditions of sequential operations. However, in the presence of interference, it makes sense to use the hooked version of read-mode variables in post conditions because these might have been modified by the environment during the execution; P-TEST$_1$ in Section 2 is a typical example. This reveals an asymmetry in the use of mode restrictions: they give commitments of the operations but no assumptions on the environment. To compensate for this, the **rd** and **wr** mode restrictions are enriched with:

- the keyword **ext** (external) if the variable can be modified by the environment;
- the keyword **ptc** (protected) if the variable can be accessed but not modified by the environment;
- the keyword **prv** (private) if the variable cannot be accessed by the environment.

The result variables of an operation are implicitly of mode **prv wr**. The use of **ext** and **ptc** mode restrictions was already advocated in [Stø91]; the novelty here is the explicit distinction between protected and private variables.

Mode restrictions are well illustrated by the decomposition of F-EQUATE$_1$ in Figure 1 (Section 2) that introduces the specifications F-ROOT$_1$ and F-TEST-AND-CONNECT$_1$. The former is used in a context where z is private and the latter is used in a context where f is protected; there are concurrent instances of ROOT in Figure 1 but the variables x and y match a **prv** mode because each of the two concurrent instances of ROOT manipulates only one of these variables.

F-ROOT$_1$

ext **rd** f : $T\text{-}forest$

prv **wr** z : T

post $F\text{-}equiv(\overleftarrow{z}, z, f) \wedge is\text{-}root(z, \overleftarrow{f})$

F-TEST-AND-CONNECT$_1$ $(c, d \colon T)\ t \colon \mathbb{B}$

ptc **wr** f : $T\text{-}forest$

guar $bodyunch(\overleftarrow{f}, f) \wedge$

$\qquad$ **let** $rest = T \setminus (F\text{-}class(c, \overleftarrow{f}) \cup F\text{-}class(d, \overleftarrow{f}))$ **in**

$\qquad$ $\forall e \in rest \cdot F\text{-}class(e, f) = F\text{-}class(e, \overleftarrow{f})$

post $(t \Leftrightarrow is\text{-}root(c, \overleftarrow{f}) \wedge is\text{-}root(d, \overleftarrow{f})) \wedge (t \Rightarrow F\text{-}equiv(c, d, f))$

With richer mode restrictions, information on interference can be better organised. First of all, only external variables have to be taken into account when the effect of environment steps has to be considered (e.g. in writing post conditions or in the proof obligations related to interference). Tool-supported proof obligations become simpler because mode restrictions identify which variables are kept unchanged by environment and/or operation steps and automatic substitution of equals simplifies proofs significantly.

Mode restrictions also play a syntactic role by restricting the set of variables whose names may occur free in the various parts of a specification. It is first observed that protected variables should not appear hooked in rely conditions because none can be modified by the environment. Private variables should not appear in the rely and guarantee conditions because those characterise visible steps and the intermediate values of private variables are invisible to other operations. As one would hope, this implies that operations on private variables have pre and post conditions only; these are indeed sequential operations and thus sequential reasoning should be the standard.

5.3 Predominance of the Post Condition

Since both the guarantee and the post condition are commitments of the operations, there can be a debate about where to put some information. For the considered class of problems (when the input/output behaviour is more important than the reactive behaviour), preference should be given to the post condition. In other words, the guarantee condition should be used for what it is intended, i.e the commitments of the operation to interference, nothing else. This policy prevents overspecification in the guarantee condition and consequently reduces the risk of unnecessary constraints on granularity. It is partially enforced by the syntactic constraints due to mode restrictions (no private variables in the guarantee condition).

5.4 Interference and Post Conditions

As the reader might have experienced in reading post conditions, these are inherently weaker and more sophisticated than their counterparts for sequential operations. A typical example is given by P-TEST$_1$ in Section 2; the same pattern (sufficient and necessary condition) can be found in F-TEST-ROOT$_1$, which is used in the termination condition of a loop in the development of F-ROOT$_1$.

F-TEST-ROOT$_1$ $(a\colon T)\ t\colon \mathbb{B}$

ext **rd** $f\ :\ T\text{-}forest$

post $(is\text{-}root(a, f)\ \Rightarrow\ t) \wedge (t\ \Rightarrow\ is\text{-}root(a, \overleftarrow{f}\,))$

Overly restrictive post conditions can make the specifications impossible to implement because of interference from other operations. Specification F-ROOT$_1$ shown earlier has a non-trivial post condition too; the naive post-condition $is\text{-}root(z, f)$ is unsatisfiable in a concurrent environment that merges trees. This mistake would be revealed by (the failure of) the most important proof obligation on specifications: the preservation of the post condition by interference.

5.5 Reasoning about Specifications

The proof that the post condition is preserved by interference creates confidence in the specification. But, as for the specifications of sequential operations, more confidence can be gained by establishing further properties of specifications. A typical check for interfering operations is to consider how the post condition simplifies in the case of less interference. For instance, the fact that z is the root of c in f easily follows from the post condition of F-ROOT$_1$ if f is not subject to interference. A less trivial example is given by the specification F-CLEANUP$_1$. If the class of a is merged with another class during its execution, CLEANUP might connect a to an element in that new class. But suppose that the equivalence class of a is preserved throughout the computation (ii); then, one may verify that the operation effectively shortens the path from a to its root (v), if possible (iii). Premise (iv) is the post condition of F-CLEANUP$_1$. This validation thus additionally shows that the evolution invariant (i) can also be thought of as a post condition.

$$
\begin{array}{rl}
(i) & a\colon T;\, f_0, f_1\colon T,\ \mathit{ev\text{-}T\text{-}forest}(f_0, f_1) \\
(ii) & \mathit{F\text{-}class}(a, f_1) = \mathit{F\text{-}class}(a, f_0) \\
(iii) & \neg\mathit{is\text{-}root}(a, f_0) \wedge \neg\mathit{is\text{-}root}(f_0(a), f_0) \\
(iv) & \neg\mathit{is\text{-}root}(a, f_0) \wedge \neg\mathit{is\text{-}root}(f_0(a), f_0) \ \Rightarrow\ f_1(a) \neq f_0(a) \\
\hline
(v) & f_1(a) \in \mathit{ancestors}(f_0(a), f_0)
\end{array}
$$

The proof is as follows:

$$
\begin{array}{rlll}
(1) & \mathit{ancestors}(a, f_1) \subseteq \mathit{F\text{-}class}(a, f_1) & \textbf{by} & (i), \text{def(s)}. \\
(2) & \mathit{F\text{-}shrink}(f_0, f_1) & \textbf{by} & (i), \text{def(s)}. \\
(3) & \mathit{ancestors}(a, f_1) \cap \mathit{F\text{-}class}(a, f_0) \subseteq \mathit{ancestors}(a, f_0) & \textbf{by} & (2), \text{def(s)}. \\
(4) & \mathit{ancestors}(a, f_1) \cap \mathit{F\text{-}class}(a, f_1) \subseteq \mathit{ancestors}(a, f_0) & \textbf{by} & (ii), (3) \\
(5) & \mathit{ancestors}(a, f_1) \subseteq \mathit{ancestors}(a, f_0) & \textbf{by} & (1), (3) \\
(6) & \neg\mathit{is\text{-}root}(a, f_1) & \textbf{by} & (iii), (2) \\
(7) & f_1(a) \in \mathit{ancestors}(a, f_1) & \textbf{by} & (i), (6), \text{def(s)}. \\
(8) & f_1(a) \in \mathit{ancestors}(a, f_0) & \textbf{by} & (5), (7) \\
(9) & f_1(a) \neq f_0(a) & \textbf{by} & (iii), (iv) \\
(10) & \mathit{ancestors}(a, f_0) = \{f_0(a)\} \cup \mathit{ancestors}(f_0(a), f_0) & \textbf{by} & (i), (iii), \text{def(s)}. \\
(v) & f_1(a) \in \mathit{ancestors}(f_0(a), f_0) & \textbf{by} & (8), (9), (10)
\end{array}
$$

All specifications make an intensive use of auxiliary functions (*ancestors*, *bodyunch*, etc). It is recommended [Jon79] to use them to develop a 'theory' of the data types involved. This not only simplifies proofs but also improves the designer's understanding of the problem.

```
protect m in
   t:= (m(x) = x ∧ m(y) = y);
   if t ∧ x ≠ y then m(x):= y endif;
end
```

Figure 2: Pseudo-code with critical sections

5.6 Transitivity

The verification that the evolution invariant and the rely conditions are transitive is another useful proof obligation. An error in the development was spotted quite late because that proof obligation had been postponed. Indeed, the evolution invariant prevents the situation where the computation of roots does not terminate because of interfering operations that, for example, first connect an element a to an element b, then connect b to a. In a preliminary development (without evolution invariant), the rely condition was

$$F\text{-}grows(\overleftarrow{f}, f) \wedge (\forall a, b: T \cdot a \in ancestors(b, \overleftarrow{f}) \;\Rightarrow\; b \notin ancestors(a, f))$$

but this fails to prevent that situation because it is not transitive; any subsequent proof using rules based on the transitivity of the rely condition is thus wrong.

6 Towards Code

The previous section was devoted to guidelines on writing specifications. How a specification is written obviously influences its subsequent development towards code but further comments can be made. Those presented in this section can only be subjective and incomplete; in particular, only comments that are specific to the treatment of interference are included.

6.1 Control over interference

As illustrated by the examples in previous sections, specification of interference is part of the design method. But not only can interference be specified; it can also be *controlled.* The search for the most adequate mechanisms to control interference in general is beyond the scope of this work but some are of course needed in the examples. This case study uses the **protect** mechanism that prevents the environment of an operation from modifying state components (no ϵ-labelled step modifies them). This mechanism is not assumed to be part of the programming language, and the decision on how to implement it has in fact been postponed.

The protected section of Figure 1 (around F-TEST-AND-CONNECT$_1$) is eventually developed into the pseudo-code of Figure 2. Protection prevents other operations from modifying m and this ensures that

1. the same m is accessed twice in the expression $m(x) = x \wedge m(y) = y$;

2. x and y are still roots in m when the connection occurs.

Nevertheless, m can still be accessed (but not modified) by other operations (e.g. TEST), even between the two accesses to m in the Boolean expression. Thus, the assignment statements in Figure 2 are not assumed to be executed atomically.

Critical sections are well known in concurrent programming (e.g. [And91]). The key issue is that such critical sections should not appear all of a sudden in the final code. They can be introduced *during the design*. This **protect** mechanism has been introduced (cf. Figure 1) in the early refinement of EQUATE(a, b) *before* the specifications F-ROOT$_1$ and F-TEST-AND-CONNECT$_1$ were further developed. Such control information is recorded by the mode restrictions introduced in Section 5 and a specification like F-TEST-AND-CONNECT$_1$ can be subsequently developed without worrying about write accesses from the environment. Mode restrictions propagate through the design to the final code: whether the occurrence of a variable in the code is protected or not follows from the design.

Control over Granularity. The **protect** mechanism does not enforce mutual exclusion in that other operations have read-access to the shared state components. If mutual exclusion (or atomic execution of an assignment statement) was required, then this should also be introduced explicitly during the design. Such a mechanism was introduced in a first attempt to implement TEST-AND-CONNECT but this appeared to be a bad design decision. Indeed, if m appears in any section where read access is forbidden, implementation of that critical section will require synchronisation overhead to be added before and after *every* access to m, including in the much executed ROOT operation.

Easiness versus efficiency. Control over interference can be necessary: roots should not be connected by other operations between the 'test' and 'connect' parts in Figure 2: protecting each part separately is not sufficient. At the other extreme, the development of EQUATE would have been easier if the whole body of the operation was under the scope of a **protect** mechanism. This would however drastically restrict concurrency! In this development, the computation of roots, which is probably the most time-consuming part of the execution of EQUATE, can be executed concurrently with any other operation.

Suppose that **protect** is implemented by a readers and writers protocol[5]. Then the only synchronisation overhead is: a reader protocol around one test in TEST (after the computation of roots), a writer protocol around the code for TEST-AND-CONNECT inside EQUATE, and a writer protocol around the only assignment statement of CLEANUP. There is no synchronisation overhead in the computation of roots.

The writer protocol around the assignment in CLEANUP is of special interest. Its presence is due to the implementation of the **protect** mechanism in other operations. This mechanism made the development of TEST-AND-CONNECT easier, but the loss of efficiency in CLEANUP seems excessive: protection is against the destruction of roots and the guarantee condition in F-CLEANUP$_1$ ensures that roots are unchanged. Thus,

[5] Details in the appendix.

on the one hand, the current formal development improves confidence in a safe removal of the synchronisation overhead in CLEANUP. But, on the other hand, it is unclear how to do it formally, in a cost-effective way.

Synchronisation and Compositionality. When the concurrent execution of several instances of EQUATE was first considered, it seemed that the addition of *explicit* synchronisation variables between the operations might be required. A fully compositional development indeed requires each operation to be developed independently down to machine code. But an attempt to add explicit synchronisation variables was quickly abandoned, first because it was unclear how to choose the variables, and second because this would have implied adding all 'protocol information' in specifications and carry all those complications through the development. An easier development that ends up with (perhaps less efficient) pseudo-code like that in Figure 2 is preferred.

6.2 Introduction of Code

As illustrated in Figure 1, language constructs (loop, ';', assignment statements) appear early in the development. This of course biases the development towards imperative programming languages, but those are the target languages, at least for the code of the individual operations. How those operations are actually activated (procedure call, message passing, etc.) is not considered in this development.

But the most interesting feature is the introduction of assignment statements. Most often, it is much easier to introduce an assignment statement than to describe it by a specification. A description of $x, y{:=}\ a, b$ in Figure 1 with guarantee and post conditions is unnecessarily opaque. A similar remark holds for the development of the elementary specifications M-CONNECT-TO-ANCESTOR$_1$ (used in CLEANUP) and M-CONNECT-ROOTS$_1$ (used in EQUATE) into the assignment statement $m(a){:=}\ b$. There is no need for any intermediate specification that would try to mimic the effect of the assignment statement in the guarantee condition. This is in accordance with the suggestion of Section 5 that the effect of an operation should be specified in the post condition rather than in the guarantee condition.

M-CONNECT-TO-ANCESTOR$_1$ $(a, b{:}\ T)$
 ext **wr** m : $T\text{-}array$
pre $b \in ancestors(a, fr(m))$
rely $bodyunch(fr(\overleftarrow{m}), fr(m))$
guar $rts(m) = rts(\overleftarrow{m}) \wedge \{a\} \ntriangleleft m = \{a\} \ntriangleleft \overleftarrow{m}$
post $m(a) = b$

M-CONNECT-ROOTS$_1$ $(a, b{:}\ T)$
 ptc **wr** m : $T\text{-}array$

pre $a \neq b \wedge a \in rts(m) \wedge b \in rts(m)$

guar $bodyunch(fr(\overleftarrow{m}), fr(m)) \wedge$

  let $rest = T \setminus (F\text{-}class(a, fr(\overleftarrow{m})) \cup F\text{-}class(b, fr(\overleftarrow{m})))$ **in**

  $\forall e \in rest \cdot F\text{-}class(e, fr(m)) = F\text{-}class(e, fr(\overleftarrow{m}))$

post $m = \overleftarrow{m} \dagger \{a \mapsto b\} \vee m = \overleftarrow{m} \dagger \{b \mapsto a\}$

The proof that the implementation of those specifications by $m(a) := b$ is correct proceeds by taking into account interference from the environment before m is assigned to; the interference after termination of the assignment statement has already been captured by the proof obligation on the post condition. Three values of m can then be identified: the initial value m_0, the value m_1 just before m is assigned to, and the value m_2 just after it is assigned to. The pre condition characterises m_0, the rely condition characterises the transitions from m_0 to m_1, and the transition from m_1 to m_2 is characterised by $m_2 = m_1 \dagger \{a \mapsto b\}$. As usual, all transitions are also characterised by the evolution invariant and $m\text{-}is\text{-}forest(m_i)$ can be assumed for each i.

6.3 Verification of the Invariants

As illustrated by M-CONNECT-TO-ANCESTOR$_2$, invariants could be expanded into the individual specifications before assignment statements are introduced.

M-CONNECT-TO-ANCESTOR$_2$ $(a, b: T)$

ptc **wr** $m : T\text{-}array$

pre $m\text{-}is\text{-}forest(m) \wedge b \in ancestors(a, fr(m))$

rely $m\text{-}is\text{-}forest(\overleftarrow{m}) \Rightarrow$

  $m\text{-}is\text{-}forest(m) \wedge ev\text{-}T\text{-}array(\overleftarrow{m}, m) \wedge bodyunch(fr(\overleftarrow{m}), fr(m))$

guar $m\text{-}is\text{-}forest(\overleftarrow{m}) \Rightarrow$

  $m\text{-}is\text{-}forest(m) \wedge ev\text{-}T\text{-}array(\overleftarrow{m}, m) \wedge rts(m) = rts(\overleftarrow{m}) \wedge \{a\} \lhd m = \{a\} \lhd m$

post $m(a) = b$

But this does not help. Keeping the invariants outside the individual specifications until code is introduced seems as easy. The preservation of invariants (between m_1 and m_2) by the assignment statement is then to be verified first. There are only two such proof obligations in this case study. The one for the refinement of M-CONNECT-TO-ANCESTOR$_1$ into $m(a) := b$ is:

$$\frac{\begin{array}{c} m_0, m_1, m_2 \colon T\text{-}array \\ m\text{-}is\text{-}forest(m_0) \wedge m\text{-}is\text{-}forest(m_1) \\ ev\text{-}T\text{-}array(m_0, m_1) \\ a \notin rts(m_0) \wedge b \in ancestors(a, fr(m_0)) \\ bodyunch(fr(m_0), fr(m_1)) \\ m_2 = m_1 \dagger \{a \mapsto b\} \end{array}}{m\text{-}is\text{-}forest(m_2) \wedge ev\text{-}T\text{-}array(m_1, m_2)}$$

Once this proof obligation is discharged, the invariants can be freely used in verifying the guarantee and post conditions of M-CONNECT-TO-ANCESTOR$_1$. Notice that a common pattern to all proofs related to assignment statements is to first show that the pre condition is preserved by interference, that is to show $b \in \mathit{ancestors}(a, \mathit{fr}(m_1))$ in this case.

7 Conclusion

Rely and guarantee conditions have been proposed to handle concurrency while preserving local reasoning in the development. Designed for the specification of interference, these conditions can also be used in an anarchic way, by encoding as much information as possible into them which quickly leads to intractable specifications. In contrast, despite the high level of concurrency, this development makes a rather economic use of rely and guarantee conditions: out of 11 specifications at the forest level, only 5 have an explicit guarantee condition, and only 3 have an explicit rely condition. A development that tends to generate many complicated rely and guarantee conditions is probably poorly organised or indicates that the specified operations fall outside the considered class of problems. In particular, the specification style in this report does not work well with operations whose reactive behaviour is the most important feature; the use of other styles of rely/guarantee specifications for the development of a non-trivial reactive system is illustrated in [KR93].

Although rely and guarantee conditions favour local reasoning, this report emphasises the role of the invariants (data invariant and evolution invariant), which by nature record global information. Therefore, local reasoning is not totally enforced because each operation is not developed independently down to code: a data reification step (with strengthening of the invariants) concerns all operations. But this is already the case for data reification steps in the development of sequential operations in VDM [Jon90] or B. The methodological importance of invariants in concurrency is not new; detailed developments based on invariants can be found –for example– in [CM88, Gri93].

As mentioned in the introduction, theoretical aspects have been intentionally neglected in the current paper. Expressiveness is one such aspect: an attentive reader should have noticed that the only restriction to concurrency in this case study is the execution of at most one instance of CLEANUP at a time. Concurrent execution of that operation not only further complicates the development but also raises expressiveness problems: it seems that the formulation of an adequate evolution invariant requires the use of history determined auxiliary variables. Use of auxiliary variables with rely/guarantee specifications is detailed in [GNL91, Stø91]. Auxiliary variables lead to clearer specifications than nested temporal operators, but inappropriate use can lead to cumbersome specifications too. At worst, rely and guarantee conditions could be reduced to an update of a history variable that records all transitions in a computation and the post condition be then expressed as a predicate on that history variable; guidelines for auxiliary variables are thus required.

The design of appropriate proof rules for data reification with rely/guarantee conditions is another theoretical aspect that deserves further work. Thanks to the evolution invariant, the problem of the appearance of new rely conditions with data reification [WD88] does not occur in this case study but might appear in others.

References

[AL93] Martin Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15:73–132, 1993.

[And91] Gregory R. Andrews. *Concurrent Programming — Principles and Practices.* Benjamin/Cummings, 1991.

[Bic92] Juan Bicarregui. Operation semantics with read and write frames. In C.B. Jones, R.C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, pages 260–278. Springer-Verlag, 1992.

[BK85] Howard Barringer and Ruud Kuiper. Hierarchical development of concurrent systems in a temporal logic framework. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 35–61. Springer-Verlag, 1985.

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

[Col94] Pierre Collette. Composition of assumption-commitment specifications in a UNITY style. *Science of Computer Programming*, 23:107–125, 1994.

[GNL91] Peter Grønning, Thomas Qvist Nielsen, and Hans Henrik Løvengreen. Refinement and composition of transition-based rely-guarantee specifications with auxiliary variables. In K.V. Nori and C.E. Veni Madhavan, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 472 of *Lecture Notes in Computer Science*, pages 332–348. Springer-Verlag, 1991.

[Gri93] Pascal Gribomont. Concurrency without toil: a systematic method for parallel program design. *Science of Computer Programming*, 21:1–56, 1993.

[Jon79] Cliff B. Jones. Constructing a theory of data structure as an aid to program development. *Acta Informatica*, 11:119–137, 1979.

[Jon81] Cliff B. Jones. *Development Methods for Computer Programs Including a Notion of Interference.* PhD thesis, Oxford University, 1981.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM.* Prentice-Hall International, Second Edition, 1990.

[JT95] Bengt Jonsson and Yih-Kuen Tsay. Assumption/guarantee specifications in linear time temporal logic. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 262–276. Springer-Verlag, 1995.

[KR93] Andrew Kay and Joy N. Reed. A rely and guarantee method for timed CSP: a specification and design of a telephone exchange. *IEEE Transactions on Software Engineering*, 19:625–639, 1993.

[MC81] Jayadev Misra and K.Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7:417–426, 1981.

[Mid93] Cornelis Middelburg. *Logic and Specification — Extending VDM-SL for Advanced Formal Specifications*. Chapman and Hall, 1993.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specifications*. Springer-Verlag, 1992.

[MV90] Caroll Morgan and Trevor Vickers. Types and invariants in the refinement calculus. *Science of Computer Programming*, 14:281–304, 1990.

[PJ91] Paritosh K. Pandya and Mathai Joseph. P-A logic — a compositional proof system for distributed programs. *Distributed Computing*, 5:27–54, 1991.

[Sta86] Eugene W. Stark. A proof technique for rely/guarantee properties. In S.N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*, pages 369–391. Springer-Verlag, 1986.

[Stø91] Ketil Stølen. An attempt to reason about shared-state concurrency in the style of VDM. In S. Prehn and W.J. Toetenel, editors, *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 324–342. Springer-Verlag, 1991.

[WD88] Jim C.P. Woodcock and B. Dickinson. Using VDM with rely and guarantee conditions. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM '88: The Way Ahead*, volume 328 of *Lecture Notes in Computer Science*, pages 434–458. Springer-Verlag, 1988.

[ZdBdR84] Job Zwiers, Arie de Bruin, and Willem-Paul de Roever. A proof system for partial correctness of dynamic networks of processes. In E. Clarke and D. Kozen, editors, *Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 513–527. Springer-Verlag, 1984.

A Technical Summary

Types and auxiliary functions

$is\text{-}disj : T\text{-}\mathbf{set} \times T\text{-}\mathbf{set} \to \mathbb{B}$

$is\text{-}disj(s_1, s_2) \quad \triangleq \quad s_1 \cap s_2 = \{\,\}$

$is\text{-}partition : (T\text{-}\mathbf{set})\text{-}\mathbf{set} \to \mathbb{B}$

$is\text{-}partition(p) \quad \triangleq \quad \bigcup p = T \wedge \{\,\} \notin p \wedge (\forall s_1, s_2 \in p \cdot s_1 = s_2 \vee is\text{-}disj(s_1, s_2))$

$$T\text{-}partition \quad = \quad \{p \in (T\text{-}\mathbf{set})\text{-}\mathbf{set} \mid is\text{-}partition(p)\}$$

$P\text{-}class : T \times T\text{-}partition \to T\text{-}\mathbf{set}$

$P\text{-}class(a, p) \quad \triangleq \quad \iota\, s \in p \cdot a \in s$

$P\text{-}equiv : T \times T \times T\text{-}partition \to \mathbb{B}$

$P\text{-}equiv(a, b, p) \quad \triangleq \quad P\text{-}class(a, p) = P\text{-}class(b, p)$

$P\text{-}grows : T\text{-}partition \times T\text{-}partition \to \mathbb{B}$

$P\text{-}grows(p_1, p_2) \quad \triangleq \quad \forall a{:}\, T \cdot P\text{-}class(a, p_1) \subseteq P\text{-}class(a, p_2)$

$in\text{-}cycles : (T \xrightarrow{m} T) \to (T\text{-}\mathbf{set})\text{-}\mathbf{set}$

$in\text{-}cycles(f) \quad \triangleq \quad \{c{:}\, T\text{-}\mathbf{set} \mid c \subseteq \mathbf{dom}\, f \wedge \forall e \in c \cdot f(e) \in c\}$

$is\text{-}forest : (T \xrightarrow{m} T) \to \mathbb{B}$

$is\text{-}forest(f) \quad \triangleq \quad in\text{-}cycles(f) = \{\,\}$

$$T\text{-}forest \quad = \quad \{f \in T \xrightarrow{m} T \mid is\text{-}forest(f)\}$$

$F\text{-}class : T \times T\text{-}forest \to T\text{-}\mathbf{set}$

$F\text{-}class(a, f) \quad \triangleq \quad \{b \mid root(b, f) = root(a, f)\}$

$F\text{-}equiv : T \times T \times T\text{-}forest \to \mathbb{B}$

$F\text{-}equiv(a, b, f) \quad \triangleq \quad F\text{-}class(a, f) = F\text{-}class(b, f)$

$is\text{-}root : T \times T\text{-}forest \to \mathbb{B}$

$is\text{-}root(a, f) \quad \triangleq \quad a \notin \mathbf{dom}\, f$

$ancestors : T \times T\text{-}forest \to T\text{-}\mathbf{set}$

$ancestors(a, f) \quad \underline{\triangle} \quad \mathbf{if}\ is\text{-}root(a, f)\ \mathbf{then}\ \{\,\}\ \mathbf{else}\ f(a) \cup ancestors(f(a), f)$

$depth : T \times T\text{-}forest \to \mathbb{N}$

$depth(a, f) \quad \underline{\triangle} \quad \mathbf{if}\ is\text{-}root(a, f)\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + depth(f(a), f)$

$F\text{-}grows : T\text{-}forest \times T\text{-}forest \to \mathbb{B}$

$F\text{-}grows(f_1, f_2) \quad \underline{\triangle} \quad \forall a : T \cdot F\text{-}class(a, f_1) \subseteq F\text{-}class(a, f_2)$

$F\text{-}shrink : T\text{-}forest \times T\text{-}forest \to \mathbb{B}$

$F\text{-}shrink(f_1, f_2) \quad \underline{\triangle} \quad \forall a : T \cdot ancestors(a, f_2) \cap F\text{-}class(a, f_1) \subseteq ancestors(a, f_1)$

$bodyunch : T\text{-}forest \times T\text{-}forest \to \mathbb{B}$

$bodyunch(f_1, f_2) \quad \underline{\triangle} \quad \forall a : T \cdot \neg is\text{-}root(a, f_1) \;\Rightarrow\; \neg is\text{-}root(a, f_2) \wedge f_2(a) = f_1(a)$

$rootunch : T\text{-}forest \times T\text{-}forest \to \mathbb{B}$

$rootunch(f_1, f_2) \quad \underline{\triangle} \quad \forall a : T \cdot is\text{-}root(a, f_1) \;\Leftrightarrow\; is\text{-}root(a, f_2)$

$$T\text{-}array \quad = \quad \{m \in T \xrightarrow{m} T \mid \mathbf{dom}\ m = T\}$$

$rts : T\text{-}array \to T\text{-}\mathbf{set}$

$rts(m) \quad \underline{\triangle} \quad \{a : T \mid m(a) = a\}$

$m\text{-}is\text{-}forest : T\text{-}array \to \mathbb{B}$

$m\text{-}is\text{-}forest(m) \quad \underline{\triangle} \quad is\text{-}forest(rts(m) \mathbin{⩤} m)$

$fr : T\text{-}array \to T\text{-}forest$

$fr(m) \quad \underline{\triangle} \quad rts(m) \mathbin{⩤} m$

$\mathbf{pre}\ m\text{-}is\text{-}forest(m)$

Development

Operation TEST:

P-TEST$_2$ $(a : T, b : T)\ t : \mathbb{B}$
$\mathbf{ext}\ \ \mathbf{rd}\ p\ :\ T\text{-}partition$
$\mathbf{post}\ (P\text{-}equiv(a, b, \overleftarrow{p}) \;\Rightarrow\; t) \wedge (t \;\Rightarrow\; P\text{-}equiv(a, b, p))$

Operation EQUATE:

P-EQUATE$_2$ $(a: T, b: T)$
ext **wr** p : $T\text{-}partition$
guar **let** $rest = T \setminus (P\text{-}class(a, \overleftarrow{p}) \cup P\text{-}class(b, \overleftarrow{p}))$ **in**
$\quad \forall e \in rest \cdot P\text{-}class(e, p) = P\text{-}class(e, \overleftarrow{p})$
post $P\text{-}equiv(a, b, p)$

Refinement of P-TEST$_2(a, b, t)$:

F-TEST$_1$ $(a: T, b: T)$ $t: \mathbb{B}$
ext **rd** f : $T\text{-}forest$
post $(F\text{-}equiv(a, b, \overleftarrow{f}) \Rightarrow t) \wedge (t \Rightarrow F\text{-}equiv(a, b, f))$

Refinement of P-EQUATE$_2(a, b)$:

F-EQUATE$_1$ $(a: T, b: T)$
ext **wr** f : $T\text{-}forest$
guar $bodyunch(\overleftarrow{f}, f) \wedge$
$\quad$ **let** $rest = T \setminus (F\text{-}class(a, \overleftarrow{f}) \cup F\text{-}class(b, \overleftarrow{f}))$ **in**
$\quad \forall e \in rest \cdot F\text{-}class(e, f) = F\text{-}class(e, \overleftarrow{f})$
post $F\text{-}equiv(a, b, f)$

Operation CLEANUP:

F-CLEANUP$_1$ $(a: T)$
ext **wr** f : $T\text{-}forest$
rely $bodyunch(\overleftarrow{f}, f)$
guar $rootunch(\overleftarrow{f}, f) \wedge \{a\} \trianglelefteq f = \{a\} \trianglelefteq \overleftarrow{f}$
post $\neg is\text{-}root(a, \overleftarrow{f}) \wedge \neg is\text{-}root(\overleftarrow{f}(a), \overleftarrow{f}) \Rightarrow f(a) \neq \overleftarrow{f}(a)$

Refinement of F-TEST$_1(a, b, t)$:

local $x, y: T$ **in**
$\quad x, y := a, b;$
$\quad$ **repeat**
$\quad\quad$ F-ROOT$_1(x) \| $F-ROOT$_1(y)$;
$\quad\quad t := (x = y)$
$\quad$ **until** $t \vee (r$ **from** (**protect** f **in** F-TEST-2-ROOTS$_1(x, y, r)$))
end

Refinement of F-EQUATE$_1(a, b)$:

local $x, y: T; t: \mathbb{B}$ **in**
$\quad x, y := a, b$
$\quad$ **repeat**
$\quad\quad$ F-ROOT$_1(x) \| $F-ROOT$_1(y)$;
$\quad\quad$ **protect** f **in** F-TEST-AND-CONNECT$_1(x, y, t)$
$\quad$ **until** t
end

F-ROOT$_1$
ext **rd** f : *T-forest*
prv **wr** z : T

post $F\text{-}equiv(\overleftarrow{z}, z, f) \wedge is\text{-}root(z, \overleftarrow{f})$

F-TEST-2-ROOTS$_1$ $(a, b\colon T)\ t\colon \mathbb{B}$
ptc **rd** f : *T-forest*

post $t \Leftrightarrow is\text{-}root(a, f) \wedge is\text{-}root(b, f)$

F-TEST-AND-CONNECT$_1$ $(c, d\colon T)\ t\colon \mathbb{B}$
ptc **rd** f : *T-forest*

guar $bodyunch(\overleftarrow{f}, f) \wedge$
  let $rest = T \setminus (F\text{-}class(c, \overleftarrow{f}) \cup F\text{-}class(d, \overleftarrow{f}))$ **in**
  $\forall e \in rest \cdot F\text{-}class(e, f) = F\text{-}class(e, \overleftarrow{f})$

post $(t \Leftrightarrow is\text{-}root(c, \overleftarrow{f}) \wedge is\text{-}root(d, \overleftarrow{f})) \wedge (t \Rightarrow F\text{-}equiv(c, d, f))$

Refinement of F-ROOT$_1(x)$:

while $\neg t$ **from** F-TEST-ROOT$_1(x, t)$
  do
  F-GO-UP$_1(x)$
  od

F-TEST-ROOT$_1$ $(a\colon T)\ t\colon \mathbb{B}$
ext **rd** f : *T-forest*

post $(is\text{-}root(a, f) \Rightarrow t) \wedge (t \Rightarrow is\text{-}root(a, \overleftarrow{f}))$

F-GO-UP$_1$
ext **rd** f : *T-forest*
prv **wr** x : T
pre $\neg is\text{-}root(x, f)$

post $F\text{-}equiv(\overleftarrow{x}, x, f) \wedge (F\text{-}equiv(\overleftarrow{x}, x, \overleftarrow{f}) \Rightarrow x \in ancestors(\overleftarrow{x}, \overleftarrow{f}))$

Refinement of F-CLEANUP$_1(a)$:

local $x\colon T$ **in**
 if $\neg t$ **from** F-TEST-ROOT$_1(a, t)$
 then F-FATHER$_1(a, x)$;
  if $\neg t$ **from** F-TEST-ROOT$_1(x, t)$
  then F-FATHER$_1(x, x)$;
   F-CONNECT-TO-ANCESTOR$_1(a, x)$
end

F-FATHER$_1$ $(a\colon T)\ x\colon T$
ext **rd** f : $T\text{-}forest$
pre $\neg is\text{-}root(a, f)$
rely $bodyunch(\overleftarrow{f}, f)$
post $x = \overleftarrow{f}(a)$

F-CONNECT-TO-ANCESTOR$_1$ $(a, b\colon T)$
ext **wr** f : $T\text{-}forest$
pre $b \in ancestors(a, f)$
rely $bodyunch(\overleftarrow{f}, f)$
guar $rootunch(\overleftarrow{f}, f) \wedge \{a\} \ntriangleleft f = \{a\} \ntriangleleft \overleftarrow{f}$
post $f(a) = b$

Refinement of F-TEST-AND-CONNECT$_1(a, b, t)$:

F-TEST-2-ROOTS$_1(a, b, t)$;
if $t \wedge a \neq b$
then F-CONNECT-ROOTS$_1(a, b)$

F-CONNECT-ROOTS$_1$ $(a\colon T, b\colon T)$
ptc **wr** f : $T\text{-}forest$
pre $a \neq b \wedge is\text{-}root(a, f) \wedge is\text{-}root(b, f)$
guar $bodyunch(\overleftarrow{f}, f) \wedge$
  let $rest = T \setminus (F\text{-}class(a, \overleftarrow{f}) \cup F\text{-}class(b, \overleftarrow{f}))$ **in**
  $\forall e \in rest \cdot F\text{-}class(e, f) = F\text{-}class(e, \overleftarrow{f})$
post $f = \overleftarrow{f} \dagger \{a \mapsto b\} \vee f = \overleftarrow{f} \dagger \{b \mapsto a\}$

Refinement of F-TEST-2-ROOTS$_1(a, b, t)$:

M-TEST-2-ROOTS$_1$ $(a, b\colon T)\ t\colon \mathbb{B}$
ptc **rd** m : $T\text{-}array$
post $t \Leftrightarrow a \in rts(m) \wedge b \in rts(m)$

Refinement of F-TEST-ROOT$_1(a, t)$:

M-TEST-ROOT$_1$ $(a\colon T)$ **wr** $t\colon \mathbb{B}$
ext **rd** m : $T\text{-}array$
post $(a \in rts(m) \Rightarrow t) \wedge (t \Rightarrow a \in rts(m))$

Refinement of F-GO-UP$_1(x)$:

M-GO-UP$_1$
ext **rd** m : $T\text{-}array$
prv **wr** x : T
pre $x \notin rts(m)$
post $F\text{-}equiv(\overleftarrow{x}, x, fr(m)) \wedge (F\text{-}equiv(\overleftarrow{x}, x, fr(\overleftarrow{m})) \Rightarrow x \in ancestors(\overleftarrow{x}, fr(\overleftarrow{m})))$

Refinement of F-FATHER$_1(a, x)$:

$$\begin{array}{l}
\text{M-FATHER}_1\ (a: T)\ x: T \\
\ \mathbf{ext}\ \ \mathbf{rd}\ m\ :\ T\text{-}array \\
\mathbf{pre}\ a \notin rts(m) \\
\mathbf{rely}\ bodyunch(fr(\overleftarrow{m}), fr(m)) \\
\mathbf{post}\ x = \overleftarrow{m}(a)
\end{array}$$

Refinement of F-CONNECT-TO-ANCESTOR$_1(a, b)$:

$$\begin{array}{l}
\text{M-CONNECT-TO-ANCESTOR}_1\ (a, b: T) \\
\ \mathbf{ext}\ \ \mathbf{wr}\ m\ :\ T\text{-}array \\
\mathbf{pre}\ b \in ancestors(a, fr(m)) \\
\mathbf{rely}\ bodyunch(fr(\overleftarrow{m}), fr(m)) \\
\mathbf{guar}\ rts(m) = rts(\overleftarrow{m}) \wedge \{a\} \lhd m = \{a\} \lhd \overleftarrow{m} \\
\mathbf{post}\ m(a) = b
\end{array}$$

Refinement of F-CONNECT-ROOTS$_1(a, b)$:

$$\begin{array}{l}
\text{M-CONNECT-ROOTS}_1\ (a, b: T) \\
\ \mathbf{ptc}\ \ \mathbf{wr}\ m\ :\ T\text{-}array \\
\mathbf{pre}\ a \neq b \wedge a \in rts(m) \wedge b \in rts(m) \\
\mathbf{guar}\ bodyunch(fr(\overleftarrow{m}), fr(m)) \wedge \\
\qquad \mathbf{let}\ rest = T \setminus (F\text{-}class(a, fr(\overleftarrow{m})) \cup F\text{-}class(b, fr(\overleftarrow{m})))\ \mathbf{in} \\
\qquad \forall e \in rest \cdot F\text{-}class(e, fr(m)) = F\text{-}class(e, fr(\overleftarrow{m})) \\
\mathbf{post}\ m = \overleftarrow{m} \dagger \{a \mapsto b\} \vee m = \overleftarrow{m} \dagger \{b \mapsto a\}
\end{array}$$

Refinement of M-TEST-2-ROOT$_1$:

$t := (m(a) = a) \wedge (m(b) = b)$

Refinement of M-TEST-ROOT$_1$:

$t := m(a) = a$

Refinement of M-GO-UP$_1$:

$x := m(x)$

Refinement of M-FATHER$_1$:

$x := m(a)$

Refinement of M-CONNECT-TO-ANCESTOR$_1$:

$m(a) := b$

Refinement of M-CONNECT-ROOTS$_1$:

$m(a) := b$

Code for the operations

```
TEST(a, b): t

local x, y: T, r: 𝔹 in
    x, y := a, b;
    repeat
        ROOT(x)||ROOT(y);
        t := (x = y);
        reader-entry-protocol
            r := (m(x) = x ∧ m(y) = y)
        reader-exit-protocol
    until t ∨ r
end
```

```
CLEANUP(a)

local x: T, t: 𝔹 in
    t := m(a) = a;
    if ¬t
    then
        x := m(a);
        t := m(x) = x;
        if ¬t
        then
            x := m(x);
            writer-entry-protocol
                m(a) := x
            writer-exit-protocol
end
```

```
EQUATE(a, b)

local x, y: T; t: 𝔹 in
    x, y := a, b;
    repeat
        ROOT(x)||ROOT(y);
        writer-entry-protocol
            t := (m(x) = x ∧ m(y) = y);
            if t ∧ x ≠ y
            then m(x) := y
        writer-exit-protocol
    until t
end
```

```
ROOT(var z)

local t: 𝔹 in
    t := m(z) = z;
    while ¬t
        do
        z := m(z);
        t := m(z) = z
        od
end
```