

Introducing Dynamic Constraints in **B**

J.-R. Abrial¹ and L. Mussat²

¹ Consultant*

26, rue des Plantes 75 014 Paris
abrial@steria.fr

² Service Central de la Sécurité des Systèmes d'Information
18, rue du Dr. Zamenhof 92 131 Issy-les-Moulineaux
ssi19@calva.net

Abstract. In **B**, the expression of dynamic constraints is notoriously missing. In this paper, we make various proposals for introducing them. They all express, in different complementary ways, how a system is allowed to evolve. Such descriptions are independent of the proposed evolutions of the system, which are defined, as usual, by means of a number of operations. Some proof obligations are thus proposed in order to reconcile the two points of view. We have been very careful to ensure that these proposals are compatible with refinement. They are illustrated by several little examples, and a larger one. In a series of small appendices, we also give some theoretical foundations to our approach. In writing this paper, we have been heavily influenced by the pioneering works of Z. Manna and A. Pnueli [11], L. Lamport [10], R. Back [5] and M. Butler [6].

1 Introduction

Background

This paper is the continuation of a study [2] that appeared in the First **B** Conference, where we showed how **B** could be used for specifying and designing distributed systems. Since then, we have explained, by means of a rather detailed case study [3], how the concepts introduced in [2] could be handled in practice and proved correct with **Atelier B** [13]. These two studies also showed that **B** could be used “as it is”, at least to a certain extent.

Previous Work

The main idea conveyed in the mentioned papers is that **B** abstract machines, normally used to specify and develop software modules, might also be used to model the evolution of certain “global” situations. For instance, the state of such a “machine” might be a complete network. And its “operations” might be simple, so called, events, which may occur “spontaneously” rather than being invoked, as is the case with a normal abstract machine operation. For instance, such an

* Supported by STERIA, SNCF, RATP and INRETS

event might be a specific action involved in a communication protocol executed by an agent that is situated somewhere in the mentioned network. Rather than being pre-conditioned, each event is guarded by a certain predicate, which states precisely the condition under which the event can be enabled. Finally, new events, which were not explicitly present in an abstraction, might be introduced in a refinement. Such events are all supposed to refine the non-event (that modifies nothing) of the abstraction. The gradual introduction of new events in successive refinement steps makes it possible to slowly develop a system by starting from a non-distributed abstraction (describing, say, in one shot, the main goal of a certain protocol) and ending eventually in a completely distributed concrete realization.

Purpose of Paper

As we became more and more familiar with the problems raised by the specification and design of such distributed systems, it clearly appeared that some of their requirements were sometimes elegantly presented informally in terms of certain “dynamic constraints” that are rather difficult to express, prove and refine with **B** “as it is”. The purpose of this paper is thus the following: (1) to present some simple extensions to **B** able to handle such constraints, (2) to show them at work on some examples, (3) to present their theoretical foundations. In doing so, we shall always keep in mind that such extensions must be easily implementable on **Atelier B**.

Dynamic Constraints

Dynamic constraints have been popular among researchers for quite a long time. They have been studied intensively for many years (an excellent survey can be found in [12]). They appear in the literature under different names such as: temporal logic constraints, liveness constraints, eventuality properties, fairness, deadlock, livelock etc. Unlike a static constraint (an invariant), which expresses that a certain property involving the state variables of a system must hold when the system is in a steady state, a dynamic constraint is a law expressing *how the system is allowed to evolve*.

Handling Dynamic Constraints in B

Our idea is to practically handle dynamic constraints in **B** in exactly the same way as we have handled so far static constraints. Let us thus recall in what follows the way we handle static constraints in **B**. As we know, it is possible to express a number of static properties of a system (in the INVARIANT clause of an abstract machine), and also to define a number of named operations (in the OPERATIONS clause) whose rôle is to describe the evolution of the system (possibly in a very abstract way, which is to be refined later). Once this is done, we are required to prove that the proposed evolution of the system is compatible with the (independently proposed) static properties: these are the

so called “proof obligations”. As can be seen, our main philosophy is one of *separation of concern*. Notice that in a specification language such as **Z** [8], the invariant has *not* to be proved to be maintained, it is *automatically* incorporated in the before-after expressions defining the dynamics of the specified system.

It seems then that everything has been said, that there is no room for further properties to be defined. The purpose of this paper is to say, no, everything has not been said: in the same way as we have defined some static properties of the system independently of the operations describing its evolution, we might also define a number of dynamic properties *independently* of the proposed operations. And, as seen above in the case of the static constraints, it will then be required to prove that the proposed operations of the system are compatible with such dynamic constraints. In other words, we want to be sure that the system indeed evolves as it has been allowed to. Notice that in a specification language such as **TLA** [10] (like in **Z** for the invariant, as we have noticed above), the dynamic constraints have *not* to be proved to be satisfied, they are *automatically* incorporated in the before-after expressions defining the dynamics of the specified system.

Short Examples

A classical generic example of a dynamic constraint, which can be defined for a system, expresses that, under some conditions, which hold *now*, a certain state of affair will certainly be reached *in the future*. In a system involving the control of some requests together with the handling of the corresponding services (such a system is thus modeled with at least two events, one introducing some new request and another honoring some pending request), we might require, as a dynamic constraint, that any pending request will not be pending for ever, that it will be honored eventually. And, again, we are then required to prove that the proposed events gently handle such a dynamic constraint. For instance, we might require of a lift system that any request for a lift at a certain floor will be satisfied in the future. Likewise, we might require that an automatic maintenance system eventually succeeds in repairing what it is supposed to repair at a given moment or explain why it cannot do so. We want to exclude the possibility for the system to do nothing and never report anything.

Another classical situation is one where, in a system, we require that once a certain property holds it then holds “for ever”: that any further evolution of the system will not destroy the property in question. For instance, in a system recording historical information, we might require that a piece of data stays in it for ever once it has been entered.

2 The Proposal

None of the proposals we present in what follows is new. They are just new in **B** and, perhaps, in the way each of them is packaged and integrated in a single language. We have been inspired by the ideas of various researchers on

different systems. Certainly the work by Z. Manna and A. Pnueli [11], and that of L. Lamport [10]. But clearly, the work of M. Butler as it is presented in [6] has been fundamental to us.

Our proposal is made of five parts. (1) We first introduce a general notion of abstract system (complementary to that of abstract machine). (2) Then we present the idea of a variant in an abstract system refinement, whose rôle is to limit the possibilities for new events, introduced in that refinement, to take control for ever. (3) We then define the concept of a dynamic invariant expressing how data are allowed to evolve in an abstract system. (4) The important notion of modality is then developed, which states how the repeated occurrences of certain events can lead to certain wishful situations. (5) Finally, we give some rules concerning the problem of deadlock.

Each of these concepts will more or less be introduced in the same systematic fashion. First, we shall present the concept (sometimes with a short background) together with its linguistic representation in **B**. Then we shall give (if any) the corresponding proof rules. At this point, we might present a little example illustrating the concept and its proof rules (the proof rules will be recognizable immediately as they will be boxed in a special way). Finally, we shall explain how the concept is influenced by refinement.

The latter point is *fundamental*. We have always to keep in mind that any technique for expressing some (dynamic) properties in the development of a system has to be compatible with some further refinements. Once you have stated a certain property at some point in the construction of a system, you demand that that property is maintained in further refinements. This is certainly part of the very definition of what refining means (the abstract promises should be honored), but dually, this is also part of the semantical definition of any *wishful* property you want to express. In fact, this gives a primary criterion for considering a property: again, it should be maintained by refinement.

2.1 Abstract Systems

Description

In order to make a clear distinction between an “ordinary” abstract machine and one that is used to model a global situation driven by some events, we do not call the latter an abstract MACHINE any more, but rather an abstract SYSTEM. Likewise, the OPERATIONS clause of such a system is now called the EVENTS clause. The other clauses are left unchanged.

A Short Example

As a toy example, next is a little system with two events:

```

SYSTEM
  toy
VARIABLES
  x, y
INVARIANT
  x, y ∈ ℕ × ℕ
INITIALIZATION
  x, y := 0, 0
EVENTS
  evt_x ≜ x := x + 1 ;
  evt_y ≜ y := y + 1
END

```

No Hidden Fairness Assumptions

In such a description, there is *no hidden assumptions* concerning the firing of the events: what you get, is only what you read. And what we can read is the following: when the system moves (notice that the “willingness” of the system to move is *not* our concern), any occurrences of the two events `evt_x` and `evt_y` may take place. However, there is no implicit “fairness” assumptions saying that both events must occur infinitely often (as is the case in **UNITY** [7]). On the contrary, here it is quite possible that the event `evt_x` never occur, or occur just once, or many times, and similarly with the event `evt_y`. In fact, besides the ability of observing any of the two events, we have no extra knowledge concerning the possible sequences of observation we will be able to make.

Abstract Scheduler

If a more specific behavior is needed, *it has to be specified explicitly*. For instance, if we want to be sure that no event can occur indefinitely without letting the other occur from time to time, and vice versa, we have to “implement” this explicitly (and we shall also see in section 2.4 how to state this, so that we will be able to prove our “implementation”). Such an implementation is called an *abstract scheduler* (a variety of such abstract schedulers is discussed in [4]).

As will be seen, such an abstract scheduler can be so abstract that it *does not commit ourselves to a specific policy*. As a consequence, any kind of more specific policy can be realized in a refinement. There are various possibilities for defining such abstract schedulers, including one consisting in introducing an explicit scheduler event.

An Abstract Scheduler Example

In what follows, we propose to distribute an abstract (fair) scheduler among the two events `evt_x` and `evt_y` incrementing the state variables x and y respectively:

```

SYSTEM
  toy_with_scheduler
VARIABLES
  x, y, c, d
INVARIANT
  x, y, c, d ∈ ℕ × ℕ × ℕ × ℕ ∧ (c > 0 ∨ d > 0)
INITIALIZATION
  x, y := 0, 0 || c, d := 1, 1
EVENTS
  evt_x ≜ SELECT c > 0 THEN x, c := x + 1, c ⇔ 1 || d := 1 END ;
  evt_y ≜ SELECT d > 0 THEN y, d := y + 1, d ⇔ 1 || c := 1 END
END

```

As can be seen, the firing of the event `evt_x` depends on the *positive* value of a certain variable `c`. The value of this variable represents, in a given state, the maximum number of time `evt_x` can be enabled without `evt_y` being itself executed. Such a variable is decreased by `evt_x` and, at the same time, `evt_x` chooses non-deterministically a *positive* value for a certain variable `d` handled similarly by the event `evt_y`. The invariant $c > 0 \vee d > 0$ ensures that the system never deadlocks. In section 2.4, it will be shown how to express (and indeed prove) formally that each of these two events is “fair” to the other.

Abstract System and Refinement

We shall postpone the discussion on the effect of refinement on an abstract system until the next section and section 2.5 (where this question will be studied in details). Let us just say for the moment that what is “promised” by an abstract system should not be destroyed by refinement. And what is promised is a number of events which can happen freely within certain (guarded) conditions. We do not want refinement to offer less concerning the occurrence of the events in question. To take the terminology of **CSP** [9] and also that of **Action Systems** as presented by Butler in [6], we do not want to restrict in any way the *external* non-determinism that is offered by an abstract system at the top level.

2.2 Refining an Abstract System: the VARIANT Clause

Description

For refining a SYSTEM, we use a REFINEMENT component whose OPERATIONS clause is also, of course, replaced by an EVENTS clause. This is also the case for a refinement refining itself an abstract system, and so on.

Besides the events of its abstraction, such a refinement may introduce some *new* events that are not present in the abstraction. Such new events are all supposed to refine a “virtual” event of the abstraction that does nothing (`skip`). This is amply described and justified in [2] and [3] (this has already been introduced for a long time in **TLA** [10] as the “stuttering steps” and also in **Action**

Systems [5] as the “internal actions”). However, as such a “virtual” skip might, in principle, take control for ever, which is certainly not desirable, we have to assume that the corresponding new event only refines a *bounded* skip. In order to ensure this limitation, we introduce a new clause: the VARIANT clause. Such a clause contains a natural number expression. A corresponding proof obligation states that each *new* event decreases that quantity, so that the *old* events cannot be postponed for ever. Notice that the variant could also be an expression denoting a natural number sequence, in which case each new event is supposed to decrease that quantity lexicographically.

Proof Obligations

Let \mathcal{H} be a new event introduced in a refinement. Suppose that the variant expression stated in the VARIANT clause is V . We have then to prove the following (where v denotes a *fresh* variable):

$$\boxed{\begin{array}{l} V \in \mathbb{N} \\ [v := V] [\mathcal{H}] (V < v) \end{array}} \quad (\text{PO.1})$$

Such a proof has to be done, as usual, under the assumptions of the various invariants we depend on in this refinement.

A Short Example

Next is another little system, in which the unique event `evt_1` non-deterministically chooses a number x and assign it to both variables a and b :

```

SYSTEM
  another_toy_0
VARIABLES
  a, b
INVARIANT
  a ∈ ℕ ∧ b ∈ ℕ ∧ b = a
INITIALIZATION
  a, b := 0, 0
EVENTS
  evt_1 ≐ ANY x WHERE x ∈ ℕ THEN a, b := x, x END
END

```

The idea is to now refine this system as follows: the concrete event `evt_1` only assigns x to a . The variable b' , the refined version of b , is set to 0. This new version of `evt_1` is only enabled when b' is equal to a . A new event, named `evt_2`, gradually transports the content of a to b' by incrementing b' while it is strictly

smaller than a . The gluing invariant $(b' = a) \Rightarrow (b' = b)$ precisely states at which moment the abstract b and the concrete b' do agree: this is indeed when the value of b' has reached that of a .

```

REFINEMENT
  another_toy_1
REFINES
  another_toy_0
VARIABLES
  a, b'
INVARIANT
  b' ∈ (0 .. a) ∧ (b' = a ⇒ b' = b)
VARIANT
  a ⇔ b'
INITIALIZATION
  a, b' := 0, 0
EVENTS
  evt_1 ≜ ANY x WHERE x ∈ ℕ ∧ b' = a THEN a, b' := x, 0 END ;
  evt_2 ≜ SELECT b' < a THEN b' := b' + 1 END

```

It is not difficult to prove that the new version of `evt_1` refines its abstract counterpart, and that `evt_2` refines `skip`. The proof obligations concerning the `VARIANT` clause reduce to the following, which holds trivially:

$$\begin{aligned}
a \in \mathbb{N} \wedge b' \in (0 .. a) &\Rightarrow a \Leftrightarrow b' \in \mathbb{N} \\
a \in \mathbb{N} \wedge b' \in (0 .. a) \wedge b' < a &\Rightarrow a \Leftrightarrow (b' + 1) < a \Leftrightarrow b'
\end{aligned}$$

The VARIANT Clause and Refinement

Another way to look at the above proof obligation is to consider that the new event \mathcal{H} refines a generalized substitution defined by the before-after predicate $V' < V$, where V' denotes the expression V with each state variable x replaced by its after-value x' . In other words, \mathcal{H} refines a certain non-deterministic substitution that strictly decreases the quantity V . Since refinement is transitive, we can then conclude that any refinement of \mathcal{H} still refines the before-after predicate $V' < V$. In conclusion, the `VARIANT` clause is *compatible with refinement*.

2.3 Dynamic Invariant: the DYNAMICS Clause

Description

A dynamic invariant is a rule stating how certain variables are allowed to evolve. For instance, it might be the case that we are sure that a certain numerical variable cannot be decreased: this is part of what we mean by this variable. It might then be useful to state this explicitly in our specification (subjected, of course, to a proof obligation expressing that each event indeed satisfies this

property). We know that, in a sense, this is redundant with the events (hence the proof obligation) but we feel it important to express such redundancies in a specification.

Such dynamic invariants are defined in a special clause called DYNAMICS. It can be introduced in an abstract system or in any of its refinements. Such a clause contains a number of conjuncted before-after predicates involving certain (before) variables and also some “primed” (after) variables. Each of these before-after predicates expresses how variables are allowed to evolve when modified by the events. For instance, given a natural number variable x , the dynamic invariant $x \leq x'$ states that the variable x is not allowed to decrease, only to increase or stay unchanged.

Proof Obligation

Of course, a proof obligation has to be generated in order to ensure that each event is indeed consistent with the contents of the dynamic invariant. For instance, an event with the substitution $x := x + 1$ satisfies the dynamic invariant $x \leq x'$ while the substitution $x := x \leftrightarrow 1$ does not.

In general, given an event \mathcal{E} and a dynamic invariant of the form $P(x, x')$ (where x denotes the variables of the considered abstract system), the shape of the proof obligation is the following, where $\text{prd}_x(\mathcal{E})$ is the before-after predicate associated with the event \mathcal{E} (this is described in the B-Book [1] in section 6.3.1):

$$\boxed{\text{prd}_x(\mathcal{E}) \Rightarrow P(x, x')} \quad (\text{PO}_2)$$

This has to be proved under the invariant of the abstract system. It exactly says what is meant by the dynamic invariant. Namely, if x and x' are related by the before-after predicate corresponding to the event \mathcal{E} then the dynamic invariant should also hold between these variables.

A Short Example

In what follows it is shown how the system *toy* of section 2.1 can be equipped with a dynamic invariant.

```

SYSTEM
  toy_with_dynamics
VARIABLES
  x, y
INVARIANT
  x, y ∈ ℕ × ℕ
DYNAMICS
  x ≤ x' ∧ y ≤ y'
INITIALIZATION
  x, y := 0, 0
EVENTS
  evt_x ≙ x := x + 1 ;
  evt_y ≙ y := y + 1
END

```

As, clearly, the before-after predicate of `evt_x` is $x' = x + 1 \wedge y' = y$, and that of `evt_y` is $y' = y + 1 \wedge x' = x$, we are led to prove the following which holds trivially:

$$\begin{aligned}
 x, y \in \mathbb{N} \times \mathbb{N} \wedge x' = x + 1 \wedge y' = y &\Rightarrow x \leq x' \wedge y \leq y' \\
 x, y \in \mathbb{N} \times \mathbb{N} \wedge y' = y + 1 \wedge x' = x &\Rightarrow x \leq x' \wedge y \leq y'
 \end{aligned}$$

Dynamic Invariant and Refinement

As for the case of the `VARIANT` clause in the preceding section, we can consider that, in the proof obligation (PO₂), the event \mathcal{E} *refines* a generalized substitution defined by the before-after predicate $P(x, x')$. And, again, since refinement is transitive, we can then conclude that any refinement of \mathcal{E} will still refine the before-after predicate $P(x, x')$. In conclusion the `DYNAMICS` clause is *compatible with refinement*.

2.4 Modality Properties: the MODALITIES Clause.

Background

A variety of temporal operators have been introduced in the literature [11] for reasoning about the way systems can behave dynamically. The most classical are \square (always), \diamond (eventually), \leadsto (leadsto) and until. In what follows, we shall give a brief informal description of each of them.

Given a certain predicate P defined on the state variables of an evolving system, then $\square P$ means that P always holds whatever the evolution of the system. In other words, this clearly means that P is invariant in our terminology.

Given a certain predicate P , the statement $\diamond P$ means that P holds at system start up or that it will certainly hold after system start up whatever the evolution of the system. Notice that, once it holds, there is no guarantee that P holds for ever. This operator is rarely used on its own, but rather together with the

operator \square , as in the statement $\square \diamond P$. This statement means that it is always the case that P holds eventually: in other words, if P does not hold in a certain state then it will certainly hold in some other state that is reachable from the former whatever the evolution of the system.

Given two predicates P and Q , the statement $P \rightsquigarrow Q$ (pronounced P leads to Q) means that it is always the case that once P holds then Q holds eventually. Notice that, once the very process, by which some progress is made towards Q , has started, then P is not required to hold any more (P is only required to hold at the initialization of this process). The statement $P \rightsquigarrow Q$ can be defined by the following combination of the two previous operators:

$$P \rightsquigarrow Q \quad \hat{=} \quad \square(P \Rightarrow \diamond Q)$$

As can be seen, the statement $\square \diamond Q$ is clearly a special case of $P \rightsquigarrow Q$.

Given two predicates P and Q , the statement P until Q means two different things: (1) that P leads to Q , and (2) that P holds as long as Q does not. Notice that it is not required that P still holds when Q just holds. Up to the last evolution, we can say that P is a “local invariant”, which holds during the active part of the process by which P “leads to” Q .

Introducing modalities in \mathbf{B}

It seems that a combination of the two operators \rightsquigarrow and until covers most of the “modalities” that one may encounter in practice. In both cases, as we have already mentioned, there is an implicit usage of a certain process that is effectively at work in order to *progress* from a certain situation characterized by the predicate P to another one characterized by the predicate Q .

Following what is done in **TLA** [10], we would like to make this process *explicit*. In other words, we would like to express that P leads to Q only when certain events $\mathcal{F}_1, \dots, \mathcal{F}_n$ do occur.

For instance, suppose that we would like to express that a certain event *evt* in a system cannot prevent indefinitely the other events in the system to occur. This is clearly expressed by stating that the guard G of *evt* leads to $\neg G$ and this when the only means of progress is precisely *evt*. If this is the case then any attempt for *evt* to take control for ever will fail.

The “eventuality” aspect of these operators means that the process in question *must come to an end*. Clearly this can be expressed as the termination of a certain loop whose body is made by the bounded choice of the concerned events and whose guard is the negation of the predicate we want to reach.

This view of our modality as a loop termination is supported by the small theoretical development that the interested reader can consult in the appendices.

Description: First Form

The first form of our modality involves five distinct components (all possibly defined in terms of the state variables x of our abstract system): (1) a predicate

P which is our “point of departure”, (2) a predicate Q which is our eventual “destination”, (3) a non-empty list of event names $\mathcal{F}_1, \dots, \mathcal{F}_n$ (possibly separated by the keyword OR) that are involved by the loop in progress to go from P to Q (this list is optional, when it is missing then all the events of the system are implicitly involved), (4) an invariant predicate J that must hold during the loop (this predicate is optional) and (5) a natural number expression V denoting the decreasing variant of our loop (alternatively, an expression denoting a sequence of natural numbers):

```

SELECT
  P
LEADSTO
  Q
WHILE
  F1 OR ... OR Fn
INVARIANT
  J
VARIANT
  V
END
```

This construct is, in fact, a proof obligation stipulating that a loop of the form below indeed terminates (where I stands for the invariant of our component):

```

SELECT
  P
THEN
  WHILE  $\neg Q$  DO
    CHOICE F1 OR ... OR Fn END
INVARIANT
  I  $\wedge$  J
VARIANT
  V
END
END
```

The termination of this loop, however, may not necessarily lead to a state where Q holds. This is because the loop may terminate *before* we reach that state. This is due to the fact that the events \mathcal{F}_1 to \mathcal{F}_n *have their own guards*. In other words, the effective guard of the loop is not $\neg Q$ but rather the following (in the B-Book, the construct `grd` is called `fis` in section 6.3.2):

$$\neg Q \wedge (\text{grd}(\mathcal{F}_1) \vee \dots \vee \text{grd}(\mathcal{F}_n))$$

Consequently, when the loop terminates we have

$$Q \vee (\neg \text{grd}(\mathcal{F}_1) \wedge \dots \wedge \neg \text{grd}(\mathcal{F}_n))$$

As we certainly do not want to terminate in a state where $(\neg \text{grd}(\mathcal{F}_1) \wedge \dots \wedge \neg \text{grd}(\mathcal{F}_n))$ holds whereas Q does not, we require the following condition to hold:

$$\neg Q \Rightarrow \text{grd}(\mathcal{F}_1) \vee \dots \vee \text{grd}(\mathcal{F}_n)$$

When $\neg Q$ is stronger than the disjunction of the guards then the effective guard of the loop is exactly $\neg Q$. When the loop terminates then we certainly are in a state where Q holds (this is justified in Appendix A7).

How to Achieve the until Effect

The until effect can easily be achieved by means of the following modality:

```

SELECT
  P
LEADSTO
  Q
WHILE
  F1 OR ... OR Fn
INVARIANT
  J ∧ (P ∨ Q)
VARIANT
  V
END

```

Since the involved loop is, as we know, guarded by $\neg Q$, the invariance of $P \vee Q$ means that *inside* the loop (where $\neg Q$ holds) then P holds, whereas *at the end* of the loop (where Q holds) P may not hold any more: it is then indeed the case “(1) that P leads to Q and (2) that P holds as long as Q does not”.

As a syntactic sugar, we can thus introduce the following construct to replace the special case above:

```

SELECT
  P
UNTIL
  Q
WHILE
  F1 OR ... OR Fn
INVARIANT
  J
VARIANT
  V
END

```

Proof Obligation for First Form

The following proof obligations (except the last one) constitute a direct derivation of the loop proof rules that one may find in the B-Book in section 9.2.9. Notice that the third and fourth proof obligations have to be repeated for each concerned event. Also notice that we do not have to prove that each of the event maintains the invariant I of the abstract system, since this has clearly been covered by the “standard” proof obligations. Note that the corresponding proofs have to be performed under the assumption of the invariant I . Finally note that the quantification is done on the variable z , which denotes the collection of state variables *that are modified in the loop*.

$$\begin{array}{l} P \Rightarrow J \\ P \Rightarrow \forall z \cdot (I \wedge J \Rightarrow V \in \mathbb{N}) \\ P \Rightarrow \forall z \cdot (I \wedge J \wedge \neg Q \Rightarrow [\mathcal{F}_i] J) \\ P \Rightarrow \forall z \cdot (I \wedge J \wedge \neg Q \Rightarrow [v := V] [\mathcal{F}_i] (V < v)) \\ P \Rightarrow \forall z \cdot (I \wedge J \wedge \neg Q \Rightarrow \text{grd}(\mathcal{F}_1) \vee \dots \vee \text{grd}(\mathcal{F}_n)) \end{array} \quad (\text{PO.3})$$

The first of these proof obligations expresses that the extra invariant is established by the guarding condition P . The second proof obligation expresses that under the invariant and the guard of the loop the variant is indeed a natural number (alternatively, an expression denoting a natural number sequence). The third proof obligation states that the predicate J is indeed an extra “local” invariant of the corresponding event. The fourth proof obligation states that the variant decreases under the corresponding event. The final proof obligation states that $\neg Q$ is the effective guard of our loop.

Notice the importance of the universal quantification over z (the state variables modified in the loop). This has the effect of separating the context of what is to be proved “dynamically” from the initial condition which does not always hold.

Description: Second Form

There exists a second form for our “leads to” property. It involves an initial non-deterministic choice. Clearly this second form generalizes the former.

```

ANY  $y$  WHERE
   $P$ 
LEADSTO
   $Q$ 
WHILE
   $\mathcal{F}_1$  OR  $\dots$  OR  $\mathcal{F}_n$ 
INVARIANT
   $J$ 
VARIANT
   $V$ 
END

```

And this second construct is, as the former, a proof obligation stipulating that a loop of the form below indeed terminates:

```

ANY  $y$  WHERE
   $P$ 
THEN
  WHILE  $\neg Q$  DO
    CHOICE  $\mathcal{F}_1$  OR  $\dots$  OR  $\mathcal{F}_n$  END
INVARIANT
   $I \wedge J$ 
VARIANT
   $V$ 
END
END

```

Proof Obligation for Second Form

By analogy with those obtained for the first form, next are the proof obligations for the second form of modality. As previously the third and fourth are to be repeated for each of the n concerned events. Notice again that these proofs have to be performed under the assumption of the current invariant I .

$$\begin{array}{l}
\forall y \cdot (P \Rightarrow J) \\
\forall y \cdot (P \Rightarrow \forall z \cdot (I \wedge J \Rightarrow V \in \mathbb{N})) \\
\forall y \cdot (P \Rightarrow \forall z \cdot (I \wedge J \wedge \neg Q \Rightarrow [\mathcal{F}_i] J)) \\
\forall y \cdot (P \Rightarrow \forall z \cdot (I \wedge J \wedge \neg Q \Rightarrow [v := V][\mathcal{F}_i] (V < v))) \\
\forall y \cdot (P \Rightarrow \forall z \cdot (I \wedge J \wedge \neg Q \Rightarrow \text{grd}(\mathcal{F}_1) \vee \dots \vee \text{grd}(\mathcal{F}_n)))
\end{array}
\tag{PO_4}$$

As for the previous case, the variable z is supposed to denote those variables that are modified in the loop.

A Short Example

As an example, we can now state that in the system *toy_with_scheduler* of section 2.1, the event *evt_x* (resp. *evt_y*) does not keep the control for ever. We have just to say that, provided its guard holds, then the recurrent occurrence of this event leads to a certain state where the guard does not hold any more, formally:

```
SYSTEM
  toy_with_scheduler_dynamics_and_modality
VARIABLES
  x, y, c, d
INVARIANT
  x, y, c, d ∈ ℕ × ℕ × ℕ × ℕ  ∧  (c > 0 ∨ d > 0)
DYNAMICS
  x ≤ x' ∧ y ≤ y'
INITIALIZATION
  x, y := 0, 0 || c, d := ℕ1 × ℕ1
EVENTS
  evt_x ≜ SELECT c > 0 THEN x, c := x + 1, c ⇔ 1 || d := ℕ1 END ;
  evt_y ≜ SELECT d > 0 THEN y, d := y + 1, d ⇔ 1 || c := ℕ1 END
MODALITIES
  SELECT c > 0 LEADSTO c = 0 WHILE evt_x VARIANT c END ;
  SELECT d > 0 LEADSTO d = 0 WHILE evt_y VARIANT d END
END
```

Notice that since the guard of *evt_x* is exactly $c > 0$, we trivially have $c > 0 \Rightarrow \text{grd}(\text{evt}_x)$ (and the like for *evt_y*). We can conclude, provided the corresponding proofs are done, that these two events are “fair” to each other.

Modalities and Refinement

For each of the two forms of modality we have introduced in this section, the four first proof obligations are clearly maintained by the refinements of the events: this is obvious for the first and second ones that are not concerned by the events; in the third and fourth proof obligations the events, say \mathcal{F} , are only present in subformulae of the form $[\mathcal{F}]R$ for some conditions R . Such statements are maintained by refinement according to the very definition of refinement, which says that if an abstraction \mathcal{F} is such that $[\mathcal{F}]R$ holds, and if \mathcal{G} refines \mathcal{F} then $[\mathcal{G}]R$ holds.

However, the main purpose of the proof obligations was to ensure that the considered loop terminated. The problem is that the implicit concrete loop now contains not only the refined versions of our events but also the new events introduced in the refinement and supposed to refine skip. So that the “body” of our refined implicit loop is now the following:

```
CHOICE  $\mathcal{G}_1$  OR  $\dots$  OR  $\mathcal{G}_n$  OR  $\mathcal{H}_1$  OR  $\dots$  OR  $\mathcal{H}_m$  END
```

where the \mathcal{G}_i are supposed to be the refinements of the more abstract \mathcal{F}_i , and where the \mathcal{H}_j are supposed to be the new events. Fortunately, we know that, thanks to the VARIANT clause introduced in section 2.2, the new events \mathcal{H}_j cannot take control for ever. It just then remains for us to be sure again that the loop does not terminate *before* its normal abstract end (as we know, refinement may have the effect of strengthening the guard of an event). In fact the following condition, which has been proved (this was a proof obligation)

$$\neg Q \Rightarrow \text{grd}(\mathcal{F}_1) \vee \dots \vee \text{grd}(\mathcal{F}_n)$$

is now refined by the condition (to be proved under the assumption of the “gluing” invariant of the refinement):

$$\neg Q \Rightarrow \text{grd}(\mathcal{G}_1) \vee \dots \vee \text{grd}(\mathcal{G}_n) \vee \text{grd}(\mathcal{H}_1) \vee \dots \vee \text{grd}(\mathcal{H}_m)$$

which has no reason to be true. We thus impose the following refinement condition (again, to be proved under the assumption of the “gluing” invariant of the refinement), which clearly is sufficient to ensure a correct refinement of each of the fifth proof obligations:

$$\text{grd}(\mathcal{F}_1) \vee \dots \vee \text{grd}(\mathcal{F}_n)$$

\Rightarrow

$$\text{grd}(\mathcal{G}_1) \vee \dots \vee \text{grd}(\mathcal{G}_n) \vee \text{grd}(\mathcal{H}_1) \vee \dots \vee \text{grd}(\mathcal{H}_m)$$

(PO.5)

This condition says that the concrete events \mathcal{G}_i (refining the abstract events \mathcal{F}_i) together with the new events \mathcal{H}_j do not deadlock more often than the abstract events \mathcal{F}_i . As we shall see in the next section, this condition must be maintained along the complete development. This is essentially the “progress condition” introduced in [6].

2.5 Deadlockfreeness

The Problem

At the top level, an abstract system “offers” a number of events able to occur “spontaneously” (guard permitting of course). Following the terminology used in CSP [9] and in Action Systems [5, 6], such possibilities correspond to a certain *external non-determinism* that has to be guaranteed. But, as we know, refinement (besides other effects) can strengthen guards, so that it could very well happen that an abstract event, say \mathcal{E} , offered at the top level, disappears completely in the refinement because of the following correct refinement

SELECT false THEN \mathcal{E} END

In order to prevent this to happen, it seems sufficient to impose that the abstract guard implies the concrete guard. But as the converse certainly holds (because of refinement), this implies that the abstract and concrete guards are identical (under the gluing invariant, of course). This is certainly too drastic a rule in the case where the refinement introduces some new events.

Proof Obligations

In fact, as we know, an abstract system can be refined by another one introducing *new* events. And we also know that such events cannot take control for ever because of the VARIANT clause (section 2.2). The idea is to impose a certain “progress condition” [6] allowing us to relax the above constraint on the refinement of the guard of an event. More precisely, in the case of the *first* refinement of an abstract system, the law is the following for each abstract event \mathcal{F} refined by \mathcal{G} and with the new events $\mathcal{H}_1, \dots, \mathcal{H}_n$ (this has to be proved under the assumption of the abstract and concrete gluing invariants):

$$\boxed{\text{grd}(\mathcal{F}) \Rightarrow \text{grd}(\mathcal{G}) \vee \text{grd}(\mathcal{H}_1) \vee \dots \vee \text{grd}(\mathcal{H}_n)} \quad (\text{PO.6})$$

Such a rule does not disallow the guard of \mathcal{F} to be strengthened in \mathcal{G} , it only gives a certain limitation to this strengthening: it should at least be “absorbed” by the new events. If we perform yet another refinement, we have to prove the following rule which gradually transports the progress condition along the development (again, this has to be proved under the relevant invariant assumptions):

$$\boxed{\begin{array}{l} \text{grd}(\mathcal{G}) \vee \text{grd}(\mathcal{H}_1) \vee \dots \vee \text{grd}(\mathcal{H}_n) \\ \Rightarrow \\ \text{grd}(\mathcal{M}) \vee \text{grd}(\mathcal{K}_1) \vee \dots \vee \text{grd}(\mathcal{K}_n) \vee \text{grd}(\mathcal{L}_1) \vee \dots \vee \text{grd}(\mathcal{L}_m) \end{array}} \quad (\text{PO.7})$$

where \mathcal{M} is a refinement of \mathcal{G} , each \mathcal{K}_i is a refinement of the corresponding more abstract event \mathcal{H}_i , and the \mathcal{L}_j are the new events of the refinement. A similar proof obligation has to be generated for each further refinement, and so on.

Example

It is easy to check that the above progress condition is satisfied in the case of the refinement *another_toy_1* of *another_toy_0* (section 2.2). We have in the more abstract system

$$\text{grd}(\text{evt}_1) \Leftrightarrow \exists x \cdot x \in \mathbb{N} \Leftrightarrow \text{true}$$

And in the concrete system

$$\begin{aligned}\text{grd}(\text{evt.1}) &\Leftrightarrow \exists x \cdot (x \in \mathbb{N} \wedge b' = a) \Leftrightarrow b' = a \\ \text{grd}(\text{evt.2}) &\Leftrightarrow b' < a\end{aligned}$$

The progress condition reduces to the following (which holds trivially):

$$b' \in (0..a) \Rightarrow (b' = a) \vee (b' < a)$$

Modalities Revisited

As we have just done for the guard of an abstract event, the proof obligation presented for the disjunction of the guards of events involved in a modality (section 2.4), has to be extended for any further refinement.

3 A Larger Example

In this section, we present a complete example able to illustrate the proposal made above.

3.1 Informal Presentation

The problem consists in specifying and refining an event system concerned with requests and services. An agency is supposed to offer n different services (n is thus a positive natural number). Clients of this agency are supposed to issue requests for these services. Once a request for a service is pending, that service cannot be requested again. On the other hand, a pending request cannot be pending for ever. At this stage, we do not require any specific scheduling strategy for honoring the pending requests. In other words, the choice of any future correct strategy must be left open. Again, our only requirement concerning any future implemented strategy is that there cannot be any pending request starvation. Although we do not specify a precise strategy, we nevertheless require a proof guaranteeing that a pending request will be served “some time”.

The system is then refined in two different ways, which both concerns the scheduling strategy: (1) a FIFO strategy, (2) a LIFT strategy. With the FIFO strategy a request is supposed to be honored according to its arrival: the oldest the first. With the LIFT strategy the n services are supposed to correspond to the n floors of a building. The clients are the passengers of the lift. The requests are issued by clients when they push the floor buttons within the lift, thus asking for the lift to stop at certain floors. The lift strategy is then the following: the lift does not change direction unless it has no more passenger to serve within that direction, and, of course, within a given direction, the lift serves its passengers in the natural order of the requested floors.

3.2 Developing the Specification

In what follows, we shall develop our formal treatment of the above problem by means of a gradual approach. Clearly, our final abstract system will have at least two events: one for issuing new requests, another for honoring pending requests. In a first phase however, we shall consider one event only, namely that corresponding to honoring requests. In a subsequent refinement, we shall then introduce the requesting event. And only at this stage shall we be able to state the dynamic constraint stipulating that no pending request can be pending for ever.

Phase 0

The (very abstract) state of our abstract system at this level, essentially formalizes the “history” of what has happened so far concerning the services that have been honored. This state consists in a “log”, named l_0 , recording which services have been honored and when. It is not our intention to formalize a notion of time in our future system: our point of view is just to consider that our model of the system *at this stage* consists in saying that certain services have been honored (one at a time) and that, clearly, such services have been honored “at some time”.

Invariant

Our variable l_0 is then just typed as follows

$$l_0 \in \mathbb{N} \rightarrow (1 \dots n)$$

This typing invariant must be made a little more precise by expressing that our history being the history “so far” is clearly finite. We write thus the following extra condition:

$$\text{dom}(l_0) \in \mathbb{F}(\mathbb{N})$$

If t is in the domain of l_0 , this means that the service $l_0(t)$ has been honored at “time” t . As can be seen, we have used the set of natural number to denote the results of time measurements. Clearly, we do not know what such a time measurement represents in terms of any precise output as given by a real clock (in particular, we are not interested in any time unit). Our only reason for formalizing time measurements with \mathbb{N} is a pragmatic one. By doing so, we shall be able to compare various time measurements, and thus easily express such concepts as before, after, next, and so on.

Of course, we could have removed completely any reference to the time by just recording the services in the order in which they have been honored (that is,

in the form of a finite sequence). However, as we shall see below, such a “trace” technique is not accurate enough in our case.

It is also convenient to have a second variable recording the time, named c_0 , of the “youngest” recorded service in the log (when it exists). This variable is typed as follows:

$$c_0 \in \mathbb{N}$$

together with the following invariant (notice that, in this invariant, $\max(\text{dom}(l_0))$ is well defined in the place where it appears because $\text{dom}(l_0)$ is then a finite and non-empty set):

$$l_0 \neq \emptyset \Rightarrow c_0 = \max(\text{dom}(l_0))$$

Dynamic Invariant

At this point, it is possible to write down a number of dynamic properties of our state variables, properties expressing that such variables can only be modified according to certain constraints. Such properties will secure the very “raison d’être” of these variables, namely to record the history of some “past” events. For instance, no service can ever be removed from the log l_0 , and no time nor any service can be modified either. Likewise, the variable c_0 can only be incremented: this formalizes that events are recorded in the log l_0 as soon as they effectively take place. This results in the following simple dynamic invariant:

$$\begin{array}{l} l_0 \subseteq l'_0 \\ c_0 < c'_0 \end{array}$$

Events

We are now ready to formalize our unique event, named *serve*. This event “spontaneously” records in the log l_0 that, at an *arbitrary* time t , necessarily strictly greater than c_0 however, some *arbitrary* service x has been honored.

```

serve  ≐
  ANY  $t, x$  WHERE
     $t \in \mathbb{N}$   ∧
     $x \in (1 .. n)$   ∧
     $c_0 < t$ 
  THEN
     $c_0 := t$   ||
     $l_0 := l_0 \cup \{t \mapsto x\}$ 
  END

```

What is important to mention here is that we do not know (and we are not interested in) the “cause” of this event. The only thing we say is that *it is possible to observe* that, subjected to certain conditions, such an arbitrary event does occur. Of course, the question is now: are we able to always observe that event? The answer to this question lies in the guard (the enabling condition) of our event, namely

$$\exists (t, x) \cdot (t \in \mathbb{N} \wedge x \in (1 .. n) \wedge c_0 < t)$$

As can be seen, this condition is always valid (under the invariant). This means that our event, as we expect, can always be observed. In other words, our system is always ready to honor any service. Notice that, at this point, we have not yet introduced any notion of request. This is precisely the subject of the next phase of our development where it will be clear that honoring a service is not done in a way that is as arbitrary as it appears in the present abstraction. It is not difficult to prove that this event maintains the static as well as the dynamic invariant.

Modality

It is interesting to note that, at this point, we can introduce the following modality expressing that time passes: any “future time” t (strictly greater than c_0) will eventually be a “past time” (some service will have occurred after it).

```

ANY  $t$  WHERE
   $c_0 < t$ 
LEADSTO
   $t \leq c_0$ 
VARIANT
   $\max(\{0, t \Leftrightarrow c_0\})$ 
END

```

Final System

Next is our complete SYSTEM (in subsequent phases we shall not show our components with all there clauses put together like this):

```
SYSTEM
  phase_0
CONSTANTS
  n
PROPERTIES
   $n \in \mathbb{N}_1$ 
VARIABLES
   $l_0, c_0$ 
INVARIANT
   $l_0 \in \mathbb{N} \rightarrow (1..n) \quad \wedge$ 
   $\text{dom}(l_0) \in \mathbb{F}(\mathbb{N}) \quad \wedge$ 
   $c_0 \in \mathbb{N} \quad \wedge$ 
   $l_0 \neq \emptyset \Rightarrow c_0 = \max(\text{dom}(l_0))$ 
DYNAMICS
   $l_0 \subseteq l'_0 \quad \wedge$ 
   $c_0 < c'_0$ 
INITIALIZATION
   $l_0, c_0 := \emptyset, 0$ 
EVENTS
  serve  $\hat{=}$ 
    ANY  $t, x$  WHERE
       $t \in \mathbb{N} \quad \wedge$ 
       $x \in (1..n) \quad \wedge$ 
       $c_0 < t$ 
    THEN
       $c_0 := t \quad \parallel$ 
       $l_0 := l_0 \cup \{t \mapsto x\}$ 
    END
MODALITIES
  ANY  $t$  WHERE
     $c_0 < t$ 
  LEADSTO
     $t \leq c_0$ 
  VARIANT
     $\max(\{0, t \Leftrightarrow c_0\})$ 
  END
END
```

Phase 1

The purpose of this phase is to introduce a requesting event together with some new interesting dynamic constraints.

Gluing Invariant

We first introduce two variables l_1 and c_1 that are just copies of their corresponding abstractions. For the sake of readability we introduce them explicitly, together with the following trivial gluing invariant:

$$\begin{array}{l} l_1 = l_0 \\ c_1 = c_0 \end{array}$$

Invariant

We now introduce a variable r_1 which records the various requests that have been made so far. This variable is typed as follows:

$$r_1 \in \mathbb{N} \rightarrow (1..n)$$

As for the log l_0 above, we have the extra constraints that r_1 is finite, namely:

$$\text{dom}(r_1) \in \mathbb{F}(\mathbb{N})$$

Dynamic Invariant

We also have to express, as for l_1 above, that r_1 records the history of the requests. Such an history cannot be modified, it can only be augmented. We have thus the following dynamic invariant:

$$r_1 \subseteq r'_1$$

Invariant Again

As can be seen, the type of r_1 is the same as that of l_1 . Contrary to what one might expect however, the expression $r_1(t)$ does *not* denote a certain service that has been *requested* at time t . It rather denotes the “knowledge” we suppose to have in this abstraction of the (possibly future) time where the corresponding service has been (or will be) honored. In other words r_1 is a log as is l_1 . But this log possibly already records a little of the future of l_1 : it anticipates l_1 . Of course, it seems rather strange that we can guess the future time where a pending request will be honored. We give ourselves the right to do so only because we

are still in an abstraction. But, as we shall see below, this right is precisely the least committed way by which we can express that no pending request will be pending for ever. The fact that r_1 anticipates l_1 is clearly described in the following invariant:

$$l_1 \subseteq r_1$$

As a consequence the pending requests are exactly those requests that are in $r_1 \leftrightarrow l_1$. It is time now to express that a service that is pending cannot be requested until it is honored. This is formalized very easily by requiring that the function $r_1 \leftrightarrow l_1$ is injective:

$$r_1 \leftrightarrow l_1 \in \mathbb{N} \mapsto (1 \dots n)$$

It remains now for us to express that the guessed times of service of the pending requests (if any) “belong to the future”. Taking into account that c_1 denotes the time of the youngest honored service (if any), we have thus the following extra invariant:

$$r_1 \leftrightarrow l_1 \neq \emptyset \Rightarrow c_1 < \min(\text{dom}(r_1 \leftrightarrow l_1))$$

Variant

As we shall see, we have a new event, called request. This new event must not have the possibility to be enabled for ever. It has thus to decrease a certain variant, which is, in fact, the following:

$$n \leftrightarrow \text{card}(r_1 \leftrightarrow l_1)$$

Events

We are now ready to define our request event as follows

```

request  ≐
  ANY  $t, y$  WHERE
     $t \in \mathbb{N} \leftrightarrow \text{dom}(r_1 \leftrightarrow l_1) \quad \wedge$ 
     $y \in (1 \dots n) \leftrightarrow \text{ran}(r_1 \leftrightarrow l_1) \quad \wedge$ 
     $c_1 < t$ 
  THEN
     $r_1 := r_1 \cup \{t \mapsto y\}$ 
  END

```

It is easy to prove that this event maintains the static as well as the dynamic invariant, that it refines skip, and that it decreases the above variant.

Next is the refinement of the event serve. As expected, the idea is to now deterministically honor the pending request (if any) whose guessed service time is the smallest.

```

serve  ≐
  SELECT
     $r_1 \leftrightarrow l_1 \neq \emptyset$ 
  THEN
    LET  $t$  BE
       $t = \min(\text{dom}(r_1 \leftrightarrow l_1))$ 
    IN
       $c_1 := t$  ||
       $l_1 := l_1 \cup \{t \mapsto r_1(t)\}$ 
    END
  END

```

It is interesting to compare this event with its abstraction:

```

serve  ≐
  ANY  $t, x$  WHERE
     $t \in \mathbb{N}$   ∧
     $x \in (1 .. n)$   ∧
     $c_0 < t$ 
  THEN
     $c_0 := t$  ||
     $l_0 := l_0 \cup \{t \mapsto x\}$ 
  END

```

As can be seen the refined version is now completely deterministic. Of course, the guard of the refined event must be stronger than that of the abstraction. This is certainly the case since, as we have already seen, the guard of the abstraction is always valid. The refinement proof also requires that we give some witness values for the arbitrary variables t and x in the abstraction: these are clearly $\min(\text{dom}(r_1 \leftrightarrow l_1))$ and $r_1(\min(\text{dom}(r_1 \leftrightarrow l_1)))$.

Modality

It finally remains for us to express that no pending request will be pending for ever. We have just to say that if any request for the service x is in the range of $r_1 \leftrightarrow l_1$ (is waiting to be honored) then, it will not stay there for ever under the recurrent occurrences of the two events request and serve. At some point in the future it should disappear (may be shortly, of course). This is done as follows:

<p> ANY x WHERE $x \in \text{ran}(r_1 \Leftrightarrow l_1)$ LEADSTO $x \notin \text{ran}(r_1 \Leftrightarrow l_1)$ VARIANT $[((1..n) \times \{c_1\}) \Leftarrow (r_1 \Leftrightarrow l_1)^{-1}(x) \Leftrightarrow c_1, n \Leftrightarrow \text{card}(r_1 \Leftrightarrow l_1)]$ END </p>
--

Notice that, in this modality, we have not mentioned any event since both events serve and request are concerned. Also note that our variant is lexicographic. The apparent complexity of the first part of this variant is due to the fact that when x is not a member of $\text{ran}(r_1 \Leftrightarrow l_1)$ (at the end of the process), the expression $(r_1 \Leftrightarrow l_1)^{-1}(x)$ is not defined. On the other hand, the second part of this variant is a mere copy of our VARIANT clause.

It is easy to prove that both events are compatible with this modality.

Deadlockfreeness

According to the outcome of section 2.5, it remains for us to ensure deadlock-freeness. In other words, we have to prove that the abstract guard of event serve implies the disjunction of the concrete guards. This reduces to proving:

$$r_1 \Leftrightarrow l_1 = \emptyset \Rightarrow \exists (t, y) \cdot (t \in \mathbb{N} \Leftrightarrow \text{dom}(r_1 \Leftrightarrow l_1) \wedge y \in (1..n) \Leftrightarrow \text{ran}(r_1 \Leftrightarrow l_1) \wedge c_1 < t)$$

Notice that this also proves that our abstract modality is correctly transported within the present refinement.

Conclusion

What we have done in this refinement was *not* to define a particular scheduler that has been, by any chance, able to satisfy our modality. By using the artifact of the “guessed” service time of a coming request, we have just been able to implement again a very *abstract scheduler*. As we shall see below, any policy that will not be contradictory with this abstraction will be a correct policy. In other words, we will not have to verify any more in the coming refinements that our modality is satisfied, we will just have to implement a concrete scheduler that refines this very general abstract one.

Phase 2

The second refinement is just a technical phase consisting in throwing away the log l_1 that has no purpose any more. The variable c_1 will thus also disappear. In fact, we are just going to keep the pending requests (corresponding to $r_1 \Leftrightarrow l_1$) but not with their guessed *service* time, only their guessed *waiting* time. This results in some drastic simplifications of the model.

Invariant

The new variable, named w_2 , is typed as follows:

$$w_2 \in (1..n) \mapsto \mathbb{N}_1$$

Notice that w_2 is injective as expected and that the waiting times are strictly positive as one would also expect.

Gluing Invariant

This variable is glued as follows to the abstraction:

$$w_2 = (r_1 \leftrightarrow l_1)^{-1} ; \text{minus}(c_1)$$

The function $\text{minus}(c_1)$ is the function that subtracts c_1 .

Events

We can now propose the following refinement of the event request:

```
request  $\hat{=}$ 
  ANY  $y, w$  WHERE
     $y \in (1..n) \leftrightarrow \text{dom}(w_2) \quad \wedge$ 
     $w \in \mathbb{N}_1 \leftrightarrow \text{ran}(w_2)$ 
  THEN
     $w_2 := w_2 \cup \{y \mapsto w\}$ 
  END
```

It is not difficult to prove that it refines its abstraction (the witness for t being clearly $c_1 + w$):

```
request  $\hat{=}$ 
  ANY  $t, y$  WHERE
     $t \in \mathbb{N} \leftrightarrow \text{dom}(r_1 \leftrightarrow l_1) \quad \wedge$ 
     $y \in (1..n) \leftrightarrow \text{ran}(r_1 \leftrightarrow l_1) \quad \wedge$ 
     $c_1 < t$ 
  THEN
     $r_1 := r_1 \cup \{t \mapsto y\}$ 
  END
```

Next is the proposed refinement of the event serve:

```

serve  ≐
  SELECT
     $w_2 \neq \emptyset$ 
  THEN
    LET  $t$  BE
       $t = \min(\text{ran}(w_2))$ 
    IN
       $w_2 := (w_2 \bowtie \{t\}) ; \text{minus}(t)$ 
    END
  END

```

This event clearly refines its abstraction:

```

serve  ≐
  SELECT
     $r_1 \leftrightarrow l_1 \neq \emptyset$ 
  THEN
    LET  $t$  BE
       $t = \min(\text{dom}(r_1 \leftrightarrow l_1))$ 
    IN
       $c_1 := t \parallel$ 
       $l_1 := l_1 \cup \{t \mapsto r_1(t)\}$ 
    END
  END

```

Deadlockfreeness

The trivial equivalence of the abstract and concrete guards of each event clearly implies the deadlockfreeness condition and the correct transportation of our modalities.

3.3 Towards some implementations

In this section, our intention is to propose two different refinements of the above abstract scheduler corresponding to the FIFO and the LIFT policies respectively.

The FIFO Policy

Invariant

In this policy we are going to reduce the inverse of the previous variable w_2 to a mere injective sequence q_3 . This yields the following typing:

$$q_3 \in \text{iseq}(1 \dots n)$$

Gluing Invariant

We have the following gluing invariant:

$$q_3 = w_2^{-1}$$

Notice that we can deduce the following assertion:

$$w_2 \neq \emptyset \Rightarrow \min(\text{ran}(w_2)) = 1$$

As can be seen, the waiting times are now all “dense” (starting at 1): they together form the domain of the sequence q_3 .

Events

The event request is now very simple (just the appending of the new request in the queue):

```
request  ≐
  ANY  $y$  WHERE
     $y \in (1..n) \Leftrightarrow \text{ran}(q_3)$ 
  THEN
     $q_3 := q_3 \leftarrow y$ 
  END
```

We can fruitfully compare it to its abstraction:

```
request  ≐
  ANY  $y, w$  WHERE
     $y \in (1..n) \Leftrightarrow \text{dom}(w_2) \quad \wedge$ 
     $w \in \mathbb{N}_1 \Leftrightarrow \text{ran}(w_2)$ 
  THEN
     $w_2 := w_2 \cup \{y \mapsto w\}$ 
  END
```

As can be seen, the guessed waiting “time” of the abstraction is now completely determined as $\text{size}(q_3) + 1$. It is precisely this determination that implements the FIFO policy together with the event serve. This event just consists in removing the first element of the queue:

```
serve    ≐
  SELECT
     $q_3 \neq []$ 
  THEN
     $q_3 := \text{tail}(q_3)$ 
  END
```

Again, it can fruitfully be compared to its abstraction:

```

serve  ≐
  SELECT
    w2 ≠ ∅
  THEN
    LET t BE
      t = min(ran(w2))
    IN
      w2 := (w2 ↘ {t}) ; minus(t)
    END
  END

```

We can see how the new version simulates its abstraction: the pair with the smallest index, 1, is removed, and 1 is indeed subtracted from the other indices (this is, in fact, exactly what `tail` does).

The LIFT Policy

Invariant

As for the FIFO policy, the LIFT policy will be a refinement of our phase 2 above. In the present case, the n services are supposed to denote the n floors of a building. We shall therefore suppose that there are at least two floors (otherwise there is no point in having a lift at all !).

$$n \geq 2$$

This phase will consist in making the waiting time function w_2 (now renamed w_4) more precise. We already know from phase 2 that it should be injective from $1 \dots n$ to \mathbb{N}_1 . We have the following trivial gluing invariant.

$$w_4 = w_2$$

We introduce to more variables. First, the lift position l_4 (this is a floor) and second the lift direction d_4 (*up* or *down*). We have thus the following typing invariant:

$$\begin{array}{l} l_4 \in 1 \dots n \\ d_4 \in \{ up, down \} \end{array}$$

We now have an extra invariant stipulating that, when it is at the bottom floor, the lift goes up, whereas it goes down, when it is at the top floor (here it helps

to have at least two floors).

$$\boxed{\begin{array}{l} l_4 = 1 \Rightarrow d_4 = up \\ l_4 = n \Rightarrow d_4 = down \end{array}}$$

Given the position and direction of the lift, our idea is to have a pre-defined function yielding the waiting time for a coming request for floor x . It corresponds to the number of floors the lift has to pass (including the last one x) in order to reach floor x . For instance, next are the two functions corresponding to the lift being at floor 4 (in a building with 7 floors) and going either down or up as indicated:

$$\begin{array}{l} f_4 = \{ 7 \mapsto 9, \\ 6 \mapsto 8, \\ 5 \mapsto 7, \\ 4 \mapsto 6, \\ 3 \mapsto 1, \\ 2 \mapsto 2, \\ 1 \mapsto 3 \} \\ \text{lift} \\ \downarrow \end{array} \qquad \begin{array}{l} g_4 = \{ 7 \mapsto 3, \\ 6 \mapsto 2, \\ 5 \mapsto 1, \\ 4 \mapsto 6, \\ 3 \mapsto 7, \\ 2 \mapsto 8, \\ 1 \mapsto 9 \} \\ \text{lift} \\ \uparrow \end{array}$$

Next are the formal definitions of these two functions (notice that they are both injective from $1..n$ to \mathbb{N}_1 as required):

$$\boxed{\begin{array}{l} f_4 \hat{=} \lambda x \cdot (x \in 1..l_4 \Leftrightarrow 1 \mid l_4 \Leftrightarrow x) \cup \lambda x \cdot (x \in l_4..n \mid l_4 + x \Leftrightarrow 2) \\ g_4 \hat{=} \lambda x \cdot (x \in l_4 + 1..n \mid x \Leftrightarrow l_4) \cup \lambda x \cdot (x \in 1..l_4 \mid 2n \Leftrightarrow (l_4 + x)) \end{array}}$$

We put them together in a function h_4 parametrized by the direction of the lift:

$$\boxed{h_4 \hat{=} \{ down \mapsto f_4, up \mapsto g_4 \}}$$

It remains for us to write our main invariant stating that w_4 is always included (because not all requests are present in general) in $h_4(d_4)$:

$$\boxed{w_4 \subseteq h_4(d_4)}$$

It is now very simple to propose the following refinement for request

$$\boxed{\begin{array}{l} \text{request} \quad \hat{=} \\ \text{ANY } y \text{ WHERE} \\ y \in (1..n) \Leftrightarrow \text{dom}(w_4) \\ \text{THEN} \\ w_4 := w_4 \cup \{ y \mapsto h_4(d_4)(y) \} \\ \text{END} \end{array}}$$

This can be fruitfully compared with its abstraction (clearly the witness for w is $h_4(d_4)(y)$):

```

request  ≐
  ANY  $y, w$  WHERE
     $y \in (1 \dots n) \Leftrightarrow \text{dom}(w_2) \quad \wedge$ 
     $w \in \mathbb{N}_1 \Leftrightarrow \text{ran}(w_2)$ 
  THEN
     $w_2 := w_2 \cup \{y \mapsto w\}$ 
  END

```

Next is the new version of the event serve

```

serve  ≐
  SELECT
     $w_4 \neq \emptyset$ 
  THEN
    LET  $t$  BE
       $t = \min(\text{ran}(w_4))$ 
    IN
      LET  $z$  BE
         $z = w_4^{-1}(t)$ 
      IN
         $w_4 := (w_4 \bowtie \{t\}) ; \text{minus}(t) \quad ||$ 
         $l_4 := z \quad ||$ 
        IF  $z = n \vee (z \neq 1 \wedge l_4 > z)$  THEN
           $d_4 := \text{down}$ 
        ELSE
           $d_4 := \text{up}$ 
        END
      END
    END
  END
END

```

It can be compared with its abstraction

```

serve  ≐
  SELECT
     $w_2 \neq \emptyset$ 
  THEN
    LET  $t$  BE
       $t = \min(\text{ran}(w_2))$ 
    IN
       $w_2 := (w_2 \bowtie \{t\}) ; \text{minus}(t)$ 
    END
  END
END

```

APPENDICES

In the following Appendices, we present a formal model of certain aspects of abstract systems. The rôle of this model is essentially to formalize the concept of “reachability” as described in the linguistic construct LEADSTO introduced in section 2.4 of the main text. In doing so, we shall be able to justify the proof obligations presented in the paper.

In Appendix **A1**, we recall the definition and main properties of conjunctive set transformers. In Appendix **A2**, we quickly present and define the main classical operators applicable to conjunctive set transformers. We recall that these operators maintain conjunctivity. In Appendix **A3**, we remind the definitions and main properties of the fixpoints of conjunctive set transformers. This allows us to introduce (in Appendix **A4**) more operators dealing with the concept of iteration. We show that these operators also maintain conjunctivity. We indicate how the least fixpoint of a conjunctive set transformer is connected with the “termination” of the iteration. In Appendices **A5** and **A6** we give more properties of the least fixpoint, properties supporting this notion of termination. In Appendix **A7** we define the notion of reachability of a certain set. This is based on the termination (fixpoint) of a certain conjunctive set transformer. In Appendix **A8**, we study how the “classical” refinement influences reachability. In Appendix **A9**, we study the main assumptions concerning the new events that may be introduced in the refinement of an abstract system. Finally, in Appendix **A10**, our study culminates with the refinement of reachability *in the presence of new events*.

A1. Conjunctive Set Transformers

The events of an abstract system are formally described by means of conjunctive set transformers (see B-Book in section 6.4.2). Let F be such a set transformer built on a set s (this is the “state” of our abstract system). We have thus:

$$F \in \mathbb{P}(s) \rightarrow \mathbb{P}(s) \tag{1}$$

In what follows, we shall make the frequent abuse of language consisting in identifying an event with its set transformer. Given a subset p of s , the set $F(p)$ denotes, as we know, the largest subset of s in which we have to be, in order for the “event” formalized by F to “terminate” in a state belonging to p .

The conjunctivity property of our set transformer F is defined as follows for each non-empty set σ of subsets of s :

$$F\left(\bigcap_{t \in \sigma} t\right) = \bigcap_{t \in \sigma} F(t) \tag{2}$$

When specialized to a set σ with two elements, the above condition reduces to the following where p and q are two subsets of s :

$$F(p \cap q) = F(p) \cap F(q) \tag{3}$$

This means that if we want to terminate within both sets p and q , we better start from within both sets $F(p)$ and $F(q)$. A simple consequence of this property is that our set transformer is *monotone*. That is, for any two subsets p and q of s , we have

$$p \subseteq q \Rightarrow F(p) \subseteq F(q) \quad (4)$$

We now define the, so-called, termination set, $\text{pre}(F)$, of F as follows (see B-Book, property 6.4.9):

$$\text{pre}(F) \hat{=} F(s) \quad (5)$$

According to this definition, the set $\text{pre}(F)$ thus denotes the largest set in which we have to be in order for the event F to “terminate” in a state belonging to s . As “belonging to s ” is not adding any constraints on the outcome, the set $\text{pre}(F)$ then just denotes the set where we have to be for the event F to simply “terminate”. In what follows, we shall suppose (unless otherwise stated) that our termination set is always equal to s itself, that is

$$F(s) = s \quad (6)$$

We now present the, so called, before-after relation, $\text{rel}(F)$, associated with F (see B-Book, property 6.4.9). When a pair (x, x') belongs to this relation, this means that the event F is able to transform the state x into the state x' . It is defined indirectly by means of the image of a subset p of s under $\text{rel}(F)^{-1}$:

$$\text{rel}(F)^{-1}[p] \hat{=} \overline{F(\overline{p})} \quad (7)$$

In what follows we shall use the letter f to denote the relation $\text{rel}(F)$. The domain, $\text{dom}(f)$, of the relation f is called the *guard*, $\text{grd}(F)$, of the set transformer F . We have thus:

$$\text{grd}(F) = \text{dom}(f) = f^{-1}[s] = \overline{F(\emptyset)} \quad (8)$$

A2. Operations on Conjunctive Set Transformers

We now present the classical operations applicable to conjunctive set transformers. This is done in the following table, where F , F_1 , F_2 and F_z are conjunctive set transformers built on a certain set s , where p is a subset of s , and t is a set:

Identity	skip_s
Pre-conditioning	$p \mid F$
Guarding	$p \implies F$
Bounded choice	$F_1 \parallel F_2$
Unbounded choice	$\parallel_{z \in t} F_z$
Sequencing	$F_1 ; F_2$

Notice that in the B-Book the operators (\mid , \implies , etc) were supposed to work with predicates and predicate transformers (generalized substitutions). Here they are rather applied to sets and set transformers. This slight shift in the notation leads, we think, to a certain simplification of the formal presentation.

We have the following definitions yielding the values of the above set transformer constructions for a given subset q of s :

$\text{skip}_s(q)$	q
$(p \mid F)(q)$	$p \cap F(q)$
$(p \implies F)(q)$	$\bar{p} \cup F(q)$
$(F_1 \parallel F_2)(q)$	$F_1(q) \cap F_2(q)$
$(\parallel_{z \in t} F_z)(q)$	$\bigcap_{z \in t} F_z(q)$
$(F_1 ; F_2)(q)$	$F_1(F_2(q))$

It is easy to prove that each of the above operation transforms conjunctive set transformers into other conjunctive set transformers.

A3. Fixpoints of Conjunctive Set Transformers

As we shall use them in the sequel, we remind in this Appendix the definition and main properties of the fixpoints of conjunctive set transformers. Let $\text{fix}(F)$ and $\text{FIX}(F)$ be defined as follows:

$$\text{fix}(F) = \bigcap \{ p \mid p \subseteq s \wedge F(p) \subseteq p \} \quad (9)$$

$$\text{FIX}(F) = \bigcup \{ p \mid p \subseteq s \wedge p \subseteq F(p) \} \quad (10)$$

Since F is monotone according to (4), then these definitions indeed lead (Tarski) to fixpoints, that is:

$$F(\text{fix}(F)) = \text{fix}(F) \quad (11)$$

$$F(\text{FIX}(F)) = \text{FIX}(F) \quad (12)$$

From (9) and (10), we can easily deduce the following, which is valid for any subset p of s :

$$F(p) \subseteq p \Rightarrow \text{fix}(F) \subseteq p \quad (13)$$

$$p \subseteq F(p) \Rightarrow p \subseteq \text{FIX}(F) \quad (14)$$

These laws will be useful to prove certain properties involving the fixpoints. For instance, from them it is easy to prove that $\text{fix}(F)$ and $\text{FIX}(F)$ are indeed respectively the least and greatest fixpoints of F . So that we have

$$\text{fix}(F) \subseteq \text{FIX}(F) \quad (15)$$

A4. More Operations on Conjunctive Set Transformers: Iterates

We are now ready to present more operations on conjunctive set transformers. They are all dealing with some form of iteration. This concept is important in our framework since the formal behavior of an abstract system (with events) is intuitively formalized by all the possible iterations one is able to perform with these events. Such operations are introduced in the following table, where F is a conjunctive set transformer, and n is natural number:

n th iterate	F^n
Closure	F°
Opening	F^\wedge

The n th iterate is defined recursively as follows:

F^0	skip_s
F^{n+1}	$F ; F^n$

The values of the other two at the subset q of s are defined as follows:

$F^\circ(q)$	$\text{FIX}(q \mid F)$
$F^\wedge(q)$	$\text{fix}(q \mid F)$

The intuitive rationale behind these definitions will appear in the sequel. For the moment, let us just see what kind of property we can derive from these *fixpoint* definitions. Concerning F° , we deduce the following, supposed to be valid for any subset q of s :

$$\begin{aligned}
& F^\circ(q) \\
&= (q \mid F)(F^\circ(q)) \\
&= q \cap F(F^\circ(q)) \\
&= \text{skip}_s(q) \cap (F ; F^\circ)(q) \\
&= (\text{skip}_s \parallel (F ; F^\circ))(q)
\end{aligned}$$

As this development could have been performed on F^\wedge as well, we have thus:

$$F^\circ = \text{skip}_s \parallel (F ; F^\circ) \quad (16)$$

$$F^\wedge = \text{skip}_s \parallel (F ; F^\wedge) \quad (17)$$

We have obtained the classical unfolding properties. Clearly, such properties show the iteration at work. More precisely, by developing these equations, we obtain something like this:

$$F^\circ = \text{skip}_s \parallel F \parallel F^2 \parallel \dots \quad (18)$$

$$F^\wedge = \text{skip}_s \parallel F \parallel F^2 \parallel \dots \quad (19)$$

It seems then that the two are indeed the same. This would be a wrong conclusion in general: in fact the “...” are, as usual, misleading. F° , being defined by a greatest fixpoint, denotes a kind of infinite object, whereas F^\wedge contains some finiteness requirement in its definition. In fact, the conjunctivity of F allows one to prove the following:

$$F^\circ = \bigsqcup_{n \in \mathcal{N}} F^n \quad (20)$$

$$F^\wedge = \text{fix}(F) \mid F^\circ \quad (21)$$

Equality (20) is easily shown by first proving $F^\circ(q) \subseteq F^n(q)$ for any natural number n . This is done by mathematical induction. From this, $F^\circ(q) \subseteq (\bigsqcup_{n \in \mathcal{N}} F^n)(q)$ follows immediately. Then the second part of (20), namely $(\bigsqcup_{n \in \mathcal{N}} F^n)(q) \subseteq F^\circ(q)$, is proved using (14) and (2). Equality (21) is essentially Theorem 9.2.1 of the B-Book.

As can be seen on (20), F° just denotes all the possible iterations without any direct concerns about termination. On the other hand, as can be seen on (21), F^\wedge is exactly F° together with the fundamental *termination requirement* that one has to start the iterations from within $\text{fix}(F)$. Consequently, as we shall see in the next Appendix, no iteration started in $\text{fix}(F)$ *can be pursued indefinitely*: we shall necessarily *reach* some points where we cannot move further.

From the definition of the termination set of a set transformer, we deduce easily (B-Book sections 9.1.3 and 9.2.4) the following (since $F(s) = s$):

$$\text{pre}(F^n) = s \quad (22)$$

$$\text{pre}(F^\circ) = s \quad (23)$$

$$\text{pre}(F^\wedge) = \text{fix}(F) \quad (24)$$

The set $\text{fix}(F)$ thus represents the set from which one has to start in order for the iterate F^\wedge to *terminate*.

The before-after relation of this iterators can be calculated easily (B-Book sections 9.1.3 and 9.2.4). We obtain (since $F(s) = s$) the following:

$$\text{rel}(F^n) = f^n \quad (25)$$

$$\text{rel}(F^\circ) = f^* \quad (26)$$

$$\text{rel}(F^\wedge) = \overline{\text{fix}(F)} \times s \cup f^* \quad (27)$$

where f^* denotes the transitive and reflexive closure of f . Notice that in case $\text{fix}(F)$ is equal to s then $\text{rel}(F^\wedge)$ is exactly f^* .

Finally, we can prove by mathematical induction that, for each natural number n , F^n is conjunctive. From this and from (20) and (21) it follows that F° and F^\wedge are also conjunctive.

A5. A Property of the Least Fixpoint of a Conjunctive Set Transformer

We now prove that $\text{fix}(F)$ only contains *finite chains* of points related by the before-after relation f . This finiteness of the chains built within $\text{fix}(F)$ nicely supports the idea that we cannot “run” for ever by iterating F from a point of $\text{fix}(F)$. The proof is by contradiction. We suppose that there exists a non-empty subset c of $\text{fix}(F)$, that is

$$c \neq \emptyset \wedge c \subseteq \text{fix}(F) \quad (28)$$

such that each element of c participates in an infinite chain (included in c) relative to the before-after relation f . Under these assumptions, we are going to prove now that c is necessarily *empty*. This contradiction indicates that there is no such set c , thus $\text{fix}(F)$ only contains finite chains.

By definition, for any element x of c , we are sure that there is an element y of c such that x and y are related through f . Thus starting from x we can continue for ever following f while remaining in c . We have thus, by definition

$$\forall x \cdot (x \in c \Rightarrow \exists y \cdot (y \in c \wedge (x, y) \in f)) \quad (29)$$

That is, equivalently (by set contraposition and according to (29))

$$c \subseteq f^{-1}[c] \Leftrightarrow \overline{f^{-1}[c]} \subseteq \bar{c} \Leftrightarrow F(\bar{c}) \subseteq \bar{c} \quad (30)$$

Consequently, we have (according to (13))

$$\text{fix}(F) \subseteq \bar{c} \quad (31)$$

According to (28), (31) and the transitivity of set inclusion, it turns out that we have $c \subseteq \bar{c}$, thus $c = \emptyset$.

A6. Property of the Finite Chains of the Fixpoint

In this Appendix, we prove that all the finite chains of $\text{fix}(F)$ end up in $\overline{\text{dom}(f)}$. This property will be exploited in the next Appendix.

Since $\text{fix}(F)$ is a fixpoint, we have $F(\text{fix}(F)) = \text{fix}(F)$. Consequently, we also have $\text{fix}(F) \subseteq F(\text{fix}(F))$. Thus $\text{fix}(F)$ is *invariant* under F . In other words, when, from any point x of $\text{fix}(F)$, we follow the before-after relation f , we stay within $\text{fix}(F)$.

As we have seen in Appendix A5, we have no infinite chain within $\text{fix}(F)$. That is, if we start from a point x_1 of $\text{fix}(F)$, choosing *any* point x_2 of $f[\{x_1\}]$ (x_2 is thus in $\text{fix}(F)$), then *any* point x_3 in $f[\{x_2\}]$ (x_3 is thus in $\text{fix}(F)$), and so on, we necessarily reach a certain point x_n of $\text{fix}(F)$ where we cannot move further because $f[\{x_n\}]$ is empty. In other words, we *eventually* reach a point lying outside the domain of f , a point of $\overline{\text{dom}(f)}$. In conclusion, all the chains of $\text{fix}(F)$ are finite and end up in $\overline{\text{dom}(f)}$.

In order to be sure that an iteration F^\wedge *always* terminate (whatever its point of departure) and reaches eventually $\overline{\text{dom}(f)}$, it is thus necessary and sufficient

to prove that $\text{fix}(F)$ is equal to s . It is well known (see B-Book at section 9.2.8) that for proving this, it is sufficient to show that F “decreases” a certain natural number expression. More generally, it is sufficient to prove that F “decreases” a certain quantity whose value belongs to a set that is well-founded by a certain relation.

Notice that F^\wedge not only reaches $\overline{\text{dom}(f)}$ when it terminates but also the various elements obtained after executing skip_s , F , F^2 , and so on. In order to reach exactly the elements of $\overline{\text{grd}(F)}$, it is necessary to only keep those points lying within $\overline{\text{grd}(F)}$. This can be done by means of the following set transformer:

$$F^\wedge ; \overline{\text{grd}(F)} \Longrightarrow \text{skip}_s \tag{32}$$

This is just Dijkstra’s “do F od” command and also a special form of the WHILE construct introduced in the B-Book in section 9.2.1. However, as far as termination is concerned, both F^\wedge and $F^\wedge ; \overline{\text{grd}(F)} \Longrightarrow \text{skip}_s$ have the same termination set, namely $\text{fix}(F)$.

A7. Reachability of any Set

Notice that, most of the time, an abstract system, whose events are collectively formalized by a conjunctive set transformer F , *does not terminate*. In general, such systems are constructed to run for ever. Resisting to failures that would force them to stop, is even one of their main requirements. One might then ask why we insisted so much in the preceding Appendices on this question of the *termination* of the iteration F^\wedge since, clearly, the corresponding termination set, namely $\text{fix}(F)$, is in general empty. In what follows, we shall clarify this point.

Since an abstract system is supposed to run for ever, it cannot be characterized by a, so-called, *final* state that it is supposed to reach eventually. This contrasts with the classical view of a computer program supposed to deliver a certain *final result*. The validation of such a program is ensured by means of a proof establishing that the specified outcome is indeed reached. If such a program is formalized by means of a conjunctive set transformer S together with a termination set p (also called the *pre-condition* set), and if the outcome is characterized by a certain subset q , then proving $p \subseteq S(q)$ means that, provided we start the program within p , then we are sure to obtain the outcome characterized by q . In a sense, a terminating program S is entirely characterized by all the possible *permanent outcomes* (such as q) we can think of.

By analogy with a program, an ever running system might be entirely characterized by all the possible *temporary outcomes* we can think of (this is a thesis). In other words, reaching one of these outcomes is not synonymous with system stop as for a program. Such an outcome might be abandoned when the system proceeds further. But what must be proved is that such an outcome can be reached as often as possible, so that it is not the case that one is *always* outside it. Given a subset p of s (the temporary outcome in question), we study thus in this Appendix the notion of *reachability* of that set. And this is where the concept of termination will reappear.

We are interested in characterizing the subset of s , from which we have to start, in order to be certain to temporarily (but eventually) reach p by following the before-after relation f of the set transformer F defined as above. The idea is to put (just for the reasoning) a supplementary constraint on the events F in the form of the extra guard \bar{p} . We are then led to prove that the set transformer $(\bar{p} \Longrightarrow F)^\wedge$ indeed terminates. In other words, allowing the system to proceed only when it is outside p , forces it to stop (hopefully thus in a state that is within p). If we want this to be always the case then we have to prove that the set $\text{fix}(\bar{p} \Longrightarrow F)$ is equal to s . More precisely, we have to prove that the following set transformer does terminate:

$$(\bar{p} \Longrightarrow F)^\wedge ; (p \Longrightarrow \text{skip}_s) \quad (33)$$

This set transformer is exactly (see B-Book in section 9.2.1):

$$\text{WHILE } \bar{p} \text{ DO } F \text{ END} \quad (34)$$

This leads to the proof obligations (PO_3) and (PO_4) presented in section 2.4. Nothing proves however that we have reached p (we only know that we have stopped at points that are outside the guard of $\bar{p} \Longrightarrow F$). In fact, we shall prove that in order to reach p we need the extra condition: $\bar{p} \subseteq \text{dom}(f)$. We thus consider the set transformer $\bar{p} \Longrightarrow F$ whose definition is (for any subset q of s):

$$(\bar{p} \Longrightarrow F)(q) = p \cup F(q) \quad (35)$$

The guard of this set transformer can be calculated as follows:

$$\text{grd}(\bar{p} \Longrightarrow F) = \overline{(\bar{p} \Longrightarrow F)(\emptyset)} = \overline{p \cup F(\emptyset)} = \bar{p} \cap \overline{F(\emptyset)} = \bar{p} \cap \text{dom}(f)$$

As a consequence and according to what has been done above in Appendices **A5** and **A6**, the set $\text{fix}(\bar{p} \Longrightarrow F)$ denotes exactly the set of points from which one can eventually reach the set $p \cup \text{dom}(f)$ by following the before-after relation of $\bar{p} \Longrightarrow F$ (this is $\bar{p} \triangleleft f$). If we want to be certain to reach p , it is thus sufficient to require that $\text{dom}(f)$ is included in p , that is, alternatively:

$$\bar{p} \subseteq \text{dom}(f) \quad (36)$$

This corresponds to the last proof obligations of (PO_3) and (PO_4) obtained in section 2.4.

A8. Refining the Reachability Condition

In this Appendix, we shall study the problem of the refinement of the reachability studied in the preceding Appendix. We shall thus consider that the set transformer F , defined as above, is now *refined* to a certain set transformer G built on a set t , that is:

$$G \in \mathbb{P}(t) \rightarrow \mathbb{P}(t) \quad (37)$$

We also suppose that the termination set of G is trivial, that is:

$$G(t) = t \quad (38)$$

This refinement is performed by means of a certain total refinement relation r from t to s :

$$r \in t \leftrightarrow s \quad \wedge \quad \text{dom}(r) = t \quad (39)$$

The set transformer F is said to be refined by G by means of the refinement relation r when the following condition holds between their respective before-after relations f and g , and between their termination sets (see B-Book in section 11.2.4)

$$r^{-1} ; g \subseteq f ; r^{-1} \quad (40)$$

$$r^{-1}[\text{pre}(F)] \subseteq \text{pre}(G) \quad (41)$$

Notice that condition (41) holds trivially. This is because we supposed that $\text{pre}(F)$ is equal to s (this is condition (6)), thus $r^{-1}[\text{pre}(F)]$ is equal to $\text{dom}(r)$, that is t (since r is total according to (39)), which is certainly included in $\text{pre}(G)$ since $\text{pre}(G)$ is equal, by definition, to $G(t)$, which was supposed to be equal to t (this is (38)).

The set p whose reachability was studied in the previous Appendix is now transformed into the set $r^{-1}[p]$ (the image of p through r^{-1}). So that the reachability of $r^{-1}[p]$ will now be ensured within the termination set of the set transformer $\overline{(r^{-1}[p] \Rightarrow G)}^\wedge$. What remains to be proved is that the reachability of p in the abstraction indeed ensures that of $r^{-1}[p]$ in the concrete world. For this, we first have to prove that $r^{-1}[p] \Rightarrow G$ is a refinement of $\overline{p} \Rightarrow F$. Formally, we have to prove (under the conditions (38) and (39)):

$$r^{-1} ; \overline{(r^{-1}[p] \triangleleft g)} \subseteq \overline{(\overline{p} \triangleleft f)} ; r^{-1} \quad (42)$$

This proof is left to the (favorite theorem prover of the) reader. A consequence of this refinement is that the iterate $\overline{(\overline{p} \Rightarrow F)}^\wedge$ is refined to the iterate $\overline{(r^{-1}[p] \Rightarrow G)}^\wedge$ (according to the monotonicity of refinement with respect to opening, see B-Book section 11.2.4). From this, we can deduce (again B-Book section 11.2.4) that the image of the termination set of $\overline{(\overline{p} \Rightarrow F)}^\wedge$ is included into the termination set of $\overline{(r^{-1}[p] \Rightarrow G)}^\wedge$, that is

$$r^{-1}[\text{fix}(\overline{p} \Rightarrow F)] \subseteq \text{fix}(\overline{(r^{-1}[p] \Rightarrow G)}) \quad (43)$$

As we know (according to Appendix A6), the finite chains of $\text{fix}(\overline{(r^{-1}[p] \Rightarrow G)})$ all end up in the set $r^{-1}[p] \cup \overline{\text{dom}(g)}$. In order for the concrete set $r^{-1}[p]$ to be reached, we have to prove that $\overline{\text{dom}(g)}$ is included in $r^{-1}[p]$, alternatively $r^{-1}[p] \subseteq \overline{\text{dom}(g)}$. In order to ensure that this is the case, we claim that it is sufficient to have the following extra condition:

$$r[\overline{\text{dom}(g)}] \subseteq \overline{\text{dom}(f)} \quad (44)$$

It then remains for us to prove the following (notice that we have supposed condition (36) stating that the set p is indeed reached by the repeated execution of $\bar{p} \implies F$):

$$\bar{p} \subseteq \text{dom}(f) \wedge r[\overline{\text{dom}(g)}] \subseteq \overline{\text{dom}(f)} \Rightarrow r^{-1}[\bar{p}] \subseteq \text{dom}(g) \quad (45)$$

The proof of (45) is left to the (favorite theorem prover of the) reader (you will notice that the hypothesis concerning the totality of the refinement relation r is fundamental).

Condition (44) can be transformed equivalently as follows, by set contraposition:

$$\text{dom}(f) \subseteq r[\overline{\overline{\text{dom}(g)}}] \quad (46)$$

A9. Formal Hypotheses and Results concerning the New Events

In the next Appendix, we are going to study the problem of reachability of a certain set p when the set transformer F is refined to a concrete set transformer G by means of a certain refinement relation r , as it was studied in Appendix A8, except that this time we shall suppose that we have some extra events formalized by a set transformer H refining skip_s and “terminating”.

In this Appendix, we are going to formalize the relevant hypotheses concerning such new events and establish a simple result about them. More precisely, all our extra events are together formalized by means of a set transformer H defined on the set t as is G in (37):

$$H \in \mathbb{P}(t) \rightarrow \mathbb{P}(t) \quad (47)$$

We also suppose, as usual that $H(t)$ is equal to t . The before-after relation $\text{rel}(H)$ associated with H is h . Since H is supposed to refine skip_s by means of the refinement relation r , we have thus the following, as a special case of the condition (40):

$$r^{-1} ; h \subseteq r^{-1} \quad (48)$$

Moreover, we suppose that H^\wedge always “terminates”, this is formalized by stating that the fixpoint of H is exactly the set t :

$$\text{fix}(H) = t \quad (49)$$

This is ensured in the main text by the proof obligation (PO_1). From this, we shall now prove that H^\wedge indeed refines skip_s . The refinement condition to be proved (see B-Book page 520) reduces to:

$$r^{-1} ; h^* \subseteq r^{-1} \quad (50)$$

For proving this it suffices to prove the following for any natural number n (since h^* is equal to $\bigcup_{n \in \mathcal{N}} h^n$):

$$r^{-1} ; h^n \subseteq r^{-1} \quad (51)$$

This can easily be proved by mathematical induction, using (48). By extension, it is easy to prove that the set transformer $(r^{-1}[\bar{p}] \implies H)^\wedge$ also refines skip_s .

A10. Refining the Reachability Condition in the Presence of New Events

In this Appendix, we shall prove the central result of our study. It essentially says that, *as far as reachability is concerned*, a set transformer F can be “simulated” by the set transformer $G \parallel H$ where G refines F , and where the set transformer H (as described in Appendix A9) formalizes the new events refining skip_s .

We have written “simulated” rather than “refined”. In fact, the simple set transformer $\overline{r^{-1}[p]} \Longrightarrow (G \parallel H)$ does *not* refine $\overline{p} \Longrightarrow F$. What we shall prove is that $(\overline{r^{-1}[p]} \Longrightarrow (G \parallel H))^\wedge$ refines $(\overline{p} \Longrightarrow F)^\wedge$. In other words, the *repeated* “execution” of $\overline{p} \Longrightarrow F$ is refined by the repeated “execution” of $\overline{r^{-1}[p]} \Longrightarrow (G \parallel H)$. The new events, formalized by H , do not induce any spoiling side effects on the *global* behavior of our system.

Let us define F' , G' and H' as follows:

$$F' \hat{=} \overline{p} \Longrightarrow F \tag{52}$$

$$G' \hat{=} \overline{r^{-1}[p]} \Longrightarrow G \tag{53}$$

$$H' \hat{=} \overline{r^{-1}[p]} \Longrightarrow H \tag{54}$$

We have thus to prove that F'^\wedge is refined by $(G' \parallel H')^\wedge$ by means of the refinement relation r . For this, we shall first observe that clearly F'^\wedge is refined by $(H'^\wedge ; G')^\wedge ; H'^\wedge$ by means of the refinement relation r . This is because G' refines F' (Appendix A3) and H'^\wedge refines skip_s (Appendix A4) both by means of the refinement relation r , and also because of the monotonicity of refinement under the operators “;” and “ \wedge ” (see B-Book section 11.2.4). It then just remains for us to prove that $(H'^\wedge ; G')^\wedge ; H'^\wedge$ is (algorithmically) refined by $(G' \parallel H')^\wedge$, since the transitivity of refinement thus ensures that F'^\wedge is refined by $(G' \parallel H')^\wedge$. For proving this, we have to show (B-Book section 11.1.2) that for any subset a of t , we have:

$$((H'^\wedge ; G')^\wedge ; H'^\wedge)(a) \subseteq (G' \parallel H')^\wedge(a) \tag{55}$$

that is

$$(H'^\wedge ; G')^\wedge(H'^\wedge(a)) \subseteq (G' \parallel H')^\wedge(a) \tag{56}$$

that is equivalently

$$\text{fix}(H'^\wedge(a) \mid (H'^\wedge ; G')) \subseteq \text{fix}(a \mid (G' \parallel H')) \tag{57}$$

Let q be defined as follows

$$q \hat{=} \text{fix}(a \mid (G' \parallel H')) \tag{58}$$

In order to prove (57), it is sufficient to prove the following (B-Book section 3.2.2):

$$H'^\wedge(a) \cap (H'^\wedge ; G')(q) \subseteq q \tag{59}$$

that is

$$H' \wedge (a) \cap H' \wedge (G'(q)) \subseteq q \quad (60)$$

that is (since $H' \wedge$ is conjunctive))

$$H' \wedge (a \cap G'(q)) \subseteq q \quad (61)$$

For this it is sufficient to prove (again B-Book section 3.2.2)

$$a \cap G'(q) \cap H'(q) \subseteq q \quad (62)$$

which is obvious since $q = a \cap G'(q) \cap H'(q)$ according to (58).

We have eventually proved that $(\bar{p} \implies F) \wedge$ is refined by $(r^{-1}[p] \implies G \parallel H) \wedge$. As above in Appendix A8, we now have to find the condition under which the set $r^{-1}[p]$ is reached. By an argument that is very similar to the one developed in Appendix A8, this condition is an extension of condition (46)

$$\text{dom}(f) \subseteq \overline{r[\overline{\text{dom}(g) \cup \text{dom}(h)}}]} \quad (63)$$

yielding

$$\forall (x, y) \cdot ((y, x) \in r \implies (x \in \text{dom}(f) \implies y \in \text{dom}(g) \vee y \in \text{dom}(h))) \quad (64)$$

This is essentially a formal setting of the proof obligation PO_5 presented in the main text.

Acknowledgments: We thank D. Méry, L. Lamport, M. Butler, and B. Legard and his colleagues for a number of discussions and comments.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press (1996)
2. J.-R. Abrial. *Extending B Without Changing it (for Developing Distributed Systems)*. First B Conference (H. Habrias editor). Nantes (1996)
3. J.-R. Abrial and L. Mussat. *Specification and Design of a Transmission Protocol by Successive Refinements Using B*. in *Mathematical Methods in Program Development* Edited by M.Broy and B. Schieder. Springer-Verlag (1997)
4. K.R. Apt and E.-R. Olderog. *Proof Rules and Transformations Dealing with Fairness*. Science of Computer Programming (1983)
5. R.J.R. Back and R. Kurki-Suonio. *Decentralization of Process Nets with Centralized Control*. 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (1983)
6. M.J. Butler. *Stepwise Refinement of Communicating Systems*. Science of Computer Programming (1996)
7. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley (1988)
8. I.J. Hayes (editor). *Specification Case Study*. Prentice-Hall (1987)
9. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall (1985)

10. L. Lamport. *The Temporal Logic of Actions*. SRC Report 57 (1991)
11. Z. Manna and A. Pnueli. *Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Systems*. Science of Computer Programming (1984)
12. A. Udaya Shankar *An Introduction to Assertional Reasoning for Concurrent Systems*. ACM Computing Survey (1993)
13. Steria. *Atelier B Version 3.3*. (1997)