Refining Distributed Systems using the B Method

Olivier Rolland **Traian Muntean**

University of Marseille & CNRS Grenoble
Communicating and Parallel Systems Research Group
Parc Scientifique de Luminy - Case 925 F-13288 Marseille
{orolland@lim.univ-mrs.fr; muntean@imag.fr}

Abstract. This paper makes a proposal for introducing into the B Method [3] a new refinement operator dedicated to take into account distribution from the very specification level through all stepwise refinement process of B abstract machines towards actual implementations. Our operator expresses how a whole system composed of many concurrent and communicating processes can be designed by refinement. Such distributed refinements allow to progressively introducing some degree of parallelism and communications into the operations of abstract machines. The proposed operator is compatible with classical algorithmic and data refinements found in the B Method, and we show how it can be used also to derive sequential programs from distributed refinements whenever the implementation will require.

1 Introduction

The intensive use of formal methods all along the process of software development from abstract specification level to implementations, ensures partly the correction of the final code. Programs are proved by construction using successive steps of derivation of programs called refinements. Further tests or *a posteriori* proofs of the generated code are far less necessary for program adequacy to specifications.

Formalisms has been developed for the specification and design of functional behaviour, which can use approaches from languages, logics, and models theories. In this framework refinement stands for the principle of mapping abstract specifications to more specific, concrete behaviour of a system through stepwise-derived abstract machines. The rationale is that the concrete behaviour obtained implements the abstract specification and in this way refinements represent a design principles by which a given behaviour specification can be transformed into implementations. When this transformation is performed by stepwise proved rules, one can expect to obtain more deterministic behaviours which correctly implement the given specifications and therefore a posteriori validation of the implementation is less or no furthermore necessary. This approach has been successful used, for instance, by Abrial [3] in the B Method and by Back&al [10, 11] in the refinement calculus for action systems.

The B Method is in way to be successfully used in different industries for the development, through proved refinements, of critical systems where security is of paramount importance [9]. However, the integration of all those components with each other's and with those developed using standard software engineering is not proved and needs tests to ensure the correction of the whole system. Furthermore, if some of the components run concurrently, the communications must be proved only when the implementation level is reached, just because often they have not been taken into account at the specification level.

Taking communications and/or synchronisation constraints into account from the very specification phase and through every level of refinement is, therefore, completing the correct stepwise derivation process, which will insure the correctness of both code and interactions. Coordination and synchronization constraints will, then, be proved by construction.

In this paper we deal with the construction of parallel systems through *distributed refinement*. Abrial has proposed, in parallel to our work, in [4] an extension to the B Method in order to take external events into account by giving an operational interpretation of a guarded command. Lano and Dick have developed an approach to deal with concurrency by combining it with a form of temporal logic [6]. Butler and Walden have shown how refinement for the action systems, hence distributed systems, can be performed using the B Method [8]. Finally, Roscoe has defined refinement relations for systems described in CSP in several ways, depending on the semantic model of the language that is used [1]. Except the last one which is based on traces and divergences refinement, all these methods are built on top of Dijkstra's weakest preconditions calculus. They all take distribution into account in some way; here we propose to integrate distribution/communication within refinement operators.

This work is partially founded by the MATISSE-IST Project http://www.matisse.eris.qinetiq.com/

This work is based on the notion of *abstract systems*, as defined in [4]. Instead of abstract machines, which are used to specify and develop software modules in B, abstract systems model the evolution of global properties. Operations are then called "events" which may occur spontaneously rather than being invoked. Those events are no more pre-conditioned, but guarded by a predicate, which express the condition under which the event can be enabled. Thanks to time refinement, it is also possible to deal with the grain of atomicity of the events. Each refinement can highlight new time details, that is, new events, which were previously hidden under the folding of time. Such events have only to refine the substitution that does nothing, substitution skip. This way, granularity is always ideal; there is no need to consider the smallest grain of atomicity, and thus a huge number of artificial distinct events, in the initial specification. However, because the B Method is not at all modified, usual refinement operators remain unchanged. Algorithmic or data refinement should also reflect the ability of an event to be invoked as many times as its guard holds under convergence conditions and they shall also express the refinement of an event by a set of others; events, or the other way round, the merge or composition of "simultaneous" events.

In this paper we propose a new refinement operator especially designed for the derivation of concurrent processes, which allows to decompose monolithic operations of an abstract machine into a set of processes, easier to further refine . The resulting abstract machines can be seen as many processes executed in parallel. All those processes can communicate with each other in order to take coordination and synchronization into account. Of course, details from the initial formal specification can be integrated all along the development steps by mean of new processes. As soon as the implementation level is reached, our operator can be used to reassemble some of the operations, to sequentialize some processes and, possibly, to match the topology of the target distributed implementation architecture.

The rest of this paper is organised as following. In section 2, we define the new refinement operator and give its properties. We apply it in different cases in section 3 in order to state several theorems dedicated to simplify and automate, as much as possible, the derivation of both distributed and sequential programs. We translate these theorems in refinement rules in section 4. Section 5 shows that the rules of refinement are quite hard to implement that is why we provide some strategies to help the development of distributed software. All these mechanisms are illustrated by a case study in section 6.

2 Distributed refinement

2.1 Definitions

In order to define the distributed refinement operator, it is mandatory, first, to define what a distributed system and its components mean in this framework. We will consider a distributed system as a set of processes running in parallel on possibly different processors. Those processes can communicate (through message passing, shared memory, etc.) and can be synchronized with each other. Moreover, all the processes are created the very moment the system begins and can be enabled all along its evolution. They can be activated as soon as all the information they need is available. If several processes can be activated simultaneously, one will be chosen non-deterministically as in the CSP model.

On the other hand, the notions of abstract systems and events, following [4], can be expressed as follows:

- each event is atomic.
- an event can be enabled as soon as its guard is valid.

The choice of the event to enable among all those whose guard is valid is non-deterministic. During the execution of an abstract machine, a single event can not be indefinitely enabled (avoiding divergence and starvation).

It appears that processes and events can be seen as equivalent. Communications and synchronization are performed through global variables representing channels, or another exchange mechanism. We can then formalize a distributed system as following (using standard B operators):

Definition 1. A distributed system P is the opening of the bounded choice of all its processes ($P_{i \in 1..n}$).

$$P \approx (P_1 \Box P_2 \Box \ldots \Box P_n)^{\wedge}$$

One can define a refinement operator especially designed for the design of distributed systems where abstract events are no more refined by their concrete counterpart, but where the opening of the bounded choice of *all* the concrete events refines the opening of the bounded choice of all the abstract events. In

order for our operator to be compatible with both algorithmic and data refinements, we built it on top of data refinement.

Definition 2. An abstract system P is distributed refined (noted $\hat{\preceq}$) by an abstract system Q if and only if the opening of the bounded choice of all the processes of P is data refined by the opening of the bounded choice of all the processes of Q.

$$P \hat{\preceq} Q \quad \hat{=} \quad (P_1 \Box P_2 \Box \dots \Box P_n)^{\wedge} \preceq (Q_1 \Box Q_2 \Box \dots \Box Q_m)^{\wedge}$$

where $P_{i \in 1..n}$ are the operations of P and $Q_{j \in 1..m}$ are the operations of Q.

2.2 Properties

Distributed refinement has far less properties than algorithmic or data refinement as opening ($^{\wedge}$) is not distributive with respect to bounded choice. Indeed:

$$P_1 \hat{\preceq} P_2 \wedge P_3 \hat{\preceq} P_4 \not\Rightarrow P_1 \Box P_3 \hat{\preceq} P_2 \Box P_4$$

Here are the only two properties, which hold.

Property 1. Bounded choice is distributed refined by both its arguments.

$$P \Box Q \hat{\preceq} P \qquad P \Box Q \hat{\preceq} Q$$

Property 2. Given that data refinement is transitive, distributed refinement is transitive.

$$P \hat{\preceq} Q \wedge Q \hat{\preceq} R \Rightarrow P \hat{\preceq} R$$

3 Distributed refinement operators

The same way *event B* abstracts time through the creation of new events [4], we shall consider that distributed refinement abstracts communications. This way, an abstract system not only refines its existing abstract events, but also either introduces new communications that were not perceptible in the abstraction or hides communications that were not compatible with the grain of atomicity. Splitting a process reveals some communication details, grouping several processes hides communication detail.

We give here six theorems, concerning as many cases of refinement, for *creation* of new processes, *splitting/decomposing* processes in concurrent components (parallelization) and *grouping* processes (abstraction). Some of them can be applied without any additional proof, others need some prerequisites.

Each of the following subsections focuses on one operator (see Appendix for the associated proofs).

Definition 3. An process P of an abstract system is modelled as a conjunctive set transformer built on a set s, which represents the set of all the possible attainable states of the system:

$$P \in P(s) \rightarrow P(s)$$

3.1 Process creation

Following "event B" where one can add new events in a refinement if those events refine skip, distributed refinement allows to add new processes progressively all along the design in order to integrate more details of the specification and to decrease the grain of atomicity.

New events that are introduced at some point of the development process must not “take control” for ever to avoid divergence of the system (for instance such new events must then decrease some positive guard condition).

Theorem 1. Let P, Q and R be three processes. The bounded choice of P and Q is distributed refined by the bounded choice of P, Q and R if skip $\Box$ R.

$$\text{skip} \Box R \Rightarrow P \Box Q \hat{\preceq} P \Box Q \Box R$$

3.2 Grouping by sequence

The use of distributed refinement allows the conception of distributed systems, but, also, simplifies the development of sequential programs. So, we must be able to group some of our processes in order to retrieve standard programming structures. This section concerns the case of the sequence.

Theorem 2. Let P, Q and R be three processes and g and h two predicates. The bounded choice of P, Q guarded by g and R guarded by h is distributed refined by the bounded choice of P and the sequence of Q and R guarded by g if Q validates h.

$$\forall\, k \cdot (k \in s \wedge Q(k) \subseteq Q(h)) \quad \Rightarrow \quad \begin{array}{l} P \,[]\, (g \Rightarrow Q) \,[]\, (h \Rightarrow R) \\ P \,[]\, (g \Rightarrow (Q \,;\, R)) \end{array}$$

3.3 Grouping by loop

Refining an abstract machine to obtain *while*-loops is quite difficult. Within the framework of distributed refinement, this task is simplified by the intensive use of the opening operator ($^\wedge$). Theorem below allows obtaining *while*-loops, as far as some simple conditions are satisfied.

Theorem 3. Let P, Q and R be three processes and g and h two predicates. The bounded choice of P, Q and R guarded by the conjunction of g and h is distributed refined by the bounded choice of P and a process formed of the sequence of Q, the opening of R guarded by h and skip guarded by $\neg h$ if g is invariant by R and Q validates g.

$$\wedge \begin{array}{l} g \subseteq R(g) \\ \forall\, k \cdot (k \subseteq s \wedge Q(k) \subseteq Q(g)) \end{array} \quad \Rightarrow \quad \begin{array}{l} P \,[]\, Q \,[]\, (g \wedge h) \Rightarrow R \\ P \,[]\, (Q \,;(h \Rightarrow R)^\wedge \,;\, (\neg h \Rightarrow \mathsf{skip})) \end{array}$$

3.4 Grouping/Splitting by selection

The use of the bounded choice operator in the definition of distributed refinement allows us to combine it very easily with the unbounded choice operator. Hence, the following theorem.

Theorem 4. Let P, Q and R be three processes and $g(x)$, h and i three predicates such that:

$g(x) \in E \rightarrow \{\mathsf{true}, \mathsf{false}\}$

Q is the unbounded choice of an operation S_x under the condition $g(x) \wedge h$

R is the unbounded choice of an operation T_x under the condition $g(x) \wedge i$.

The bounded choice of P, Q and R is equal to the bounded choice of P and the unbounded choice under the condition $g(x)$ of the bounded choice of S_x guarded by h and T_x guarded by i.

$$= \begin{array}{l} P \,[]\, (@x \cdot (x \in E \wedge g(x) \wedge h \Rightarrow S_x)) \,[]\, (@x \cdot (x \in E \wedge g(x) \wedge i \Rightarrow T_x)) \\ P \,[]\, (@x \cdot (x \in E \wedge g(x) \Rightarrow (h \Rightarrow S_x \,[]\, i \Rightarrow T_x))) \end{array}$$

3.5 Grouping/Splitting by condition

Again, as the definition of distributed refinement integrates the bounded choice operator, we can take advantage of its associativity property. Theorem 5 deals with conditional statements.

Theorem 5. Let P, Q and R be three processes and g a predicate. The opening of the bounded choice of P, Q guarded by g and R guarded by $\neg g$ is equal to the opening of the bounded choice of P and a process S being the bounded choice of Q guarded by g and R guarded by $\neg g$.

$(P \,[]\, (g \triangleright Q) \,[]\, (\neg g \triangleright R))^\wedge = (P \,[]\, ((g \triangleright Q) \,[]\, (\neg g \triangleright Q)))^\wedge$

3.6 Splitting by iteration

All along the development process, events must be transformed (refined) as much as possible to be expressed by guarded commands. Theorem 6 below allows to eliminate an unbounded choice.

Theorem 6. Let u and v be two variables in E. Let $g(x)$ be a predicate dependent on its parameter x. Let P and Q be two processes such that Q affects to u any value of E which validates the predicate $g(u)$. The bounded choice of P and Q is distributed refined by the bounded choice of P and two processes R and S such that:

R assigns to *v* any value in *E* if *g*(*v*) does not hold
S assigns to *u* the value of *v* if *g*(*v*) holds.
(P □ u : g(u)) $\hat{\preceq}$ *(P □ Øg(v) Þ v :Î E □ g(v) Þ u := v)*

4 Extending the B Method

In this section, we will translate all the previous theorems in abstract systems and their refinements to clarify minds. It is to be noted that the B-tool does not implement distributed refinement and is not able to prove many of those refinements.

We define below seven rules of refinement and their condition of use, if any. Thus, when a developer encounters a matching abstract system, he can directly deduce the corresponding refinement with a minimum of proofs. It is assumed that, as most of our refinement rules do not have prerequisites, it is inadequate to automate the entire refinement process. On the other hand, it would be much more realistic to develop an interactive software which, when recognizing a matching abstract system, asks the user if he wants to apply a predefined refinement rule.

4.1 Process creation

By applying theorem 1, we add a new process op_Q in the refined system provided that op_Q refines the operation that does nothing (skip).

SYSTEM machine OPERATIONS $op_{P1} \hat{=}$ SELECT p_1 THEN P_1 END ... $op_{Pn} \hat{=}$ SELECT p_n THEN P_n END END	REFINEMENT refinement REFINES machine OPERATIONS $op_{P1} \hat{=}$ SELECT p_1 THEN P_1 END ... $op_{Pn} \hat{=}$ SELECT p_n THEN P_n END $op_Q \hat{=}$ SELECT q THEN Q END END

provided that skip □ op_Q

4.2 Grouping by sequence

By applying theorem 2, we sequentialize the processes op_P and op_Q. This refinement requires that the guard of op_Q be validated by op_P.

SYSTEM machine OPERATIONS $op_P \hat{=}$ SELECT p THEN P END $op_Q \hat{=}$ SELECT q THEN Q END $op_{R1} \hat{=}$ SELECT r_1 THEN R_1 END ...	REFINEMENT refinement REFINES machine OPERATIONS $op_{P,Q} \hat{=}$ SELECT p THEN P ; Q END $op_{R1} \hat{=}$ SELECT r_1 THEN R_1 END ... $op_{Rn} \hat{=}$ SELECT r_n THEN

$op_{Rn} \hat{=}$ SELECT r_n THEN R_n END END	R_n END END

provided that $[op_P]q \Leftrightarrow$ true

4.3 Grouping by loop

By applying theorem 3, we transform the two processes op_Q and op_S into the unique one $op_{Q,S}$ where we recognize a standard programming structure, a *while*-loop. Therefore, Q is the initialization statement; h is the break condition and S the body of the loop. This refinement requires that a part of the guard of op_S is invariant within op_S and this very same part is validated by op_Q.

SYSTEM machine OPERATIONS $op_Q \hat{=}$ SELECT q THEN Q END $op_R \hat{=}$ SELECT $r \wedge s$ THEN R END $op_{P1} \hat{=}$ SELECT p_1 THEN P_1 END ... $op_{Pn} \hat{=}$ SELECT p_n THEN P_n END END	REFINEMENT refinement REFINES machine OPERATIONS $op_{Q,R} \hat{=}$ SELECT q THEN Q ; WHILE s DO R END END $op_{P1} \hat{=}$ SELECT p_1 THEN P_1 END ... $op_{Pn} \hat{=}$ SELECT p_n THEN P_n END END

provided that $\begin{matrix} r & \Rightarrow & [op_R]r \\ [op_Q]r & \Leftrightarrow & \text{true} \end{matrix}$

4.4 Grouping/Splitting by selection

By applying theorem 4, we transform two processes formed of unbounded choices (built on the same data sets) into a single one.

SYSTEM machine OPERATIONS $op_S \hat{=}$ ANY x WHERE $g(x) \wedge h$ THEN S END $op_T \hat{=}$ ANY x WHERE $g(x) \wedge i$ THEN T END $op_{P1} \hat{=}$ SELECT p_1 THEN P_1 END ... $op_{Pn} \hat{=}$ SELECT p_n THEN P_n	REFINEMENT refinement REFINES machine OPERATIONS $op_{S,T} \hat{=}$ ANY x WHERE $g(x)$ THEN CHOICE SELECT h THEN S END OR SELECT i THEN T END END END $op_{P1} \hat{=}$ SELECT p_1 THEN

```
        END
END
```

```
        P1
        END
...
op_Pn ≙ SELECT p_n THEN
        P_n
        END
END
```

The reverse refinement is also correct.

```
SYSTEM
 machine
OPERATIONS
 op_S,T ≙ ANY x WHERE
          g(x)
          THEN
           CHOICE
            SELECT h THEN
             S
            END
           OR
            SELECT i THEN
             T
            END
           END
          END
 op_P1 ≙ SELECT p_1 THEN
          P_1
          END
 ...
 op_Pn ≙ SELECT p_n THEN
          P_n
          END
```

```
REFINEMENT
 refinement
REFINES
 machine
OPERATIONS
 op_S ≙ ANY x WHERE
          g(x) ∧ h
          THEN
           S
          END
 op_T ≙ ANY x WHERE
          g(x) ∧ i
          THEN
           T
          END
 op_P1 ≙ SELECT p_1 THEN
          P_1
          END
 ...
 op_Pn ≙ SELECT p_n THEN
          P_n
          END
END
```

4.5 Grouping/Splitting by condition

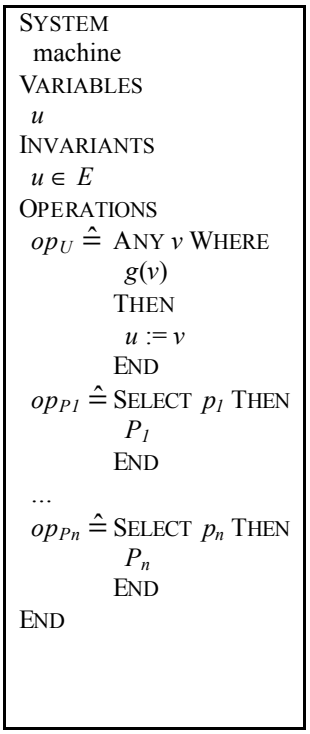
By theorem 5, we transform two processes whose guards are complementary into a conditional statement process:

```
SYSTEM
 machine
OPERATIONS
 op_S ≙ SELECT g THEN
          S
          END
 op_T ≙ SELECT ¬g THEN
          T
          END
 op_P1 ≙ SELECT p_1 THEN
          P_1
          END
 ...
 op_Pn ≙ SELECT p_n THEN
          P_n
          END
END
```

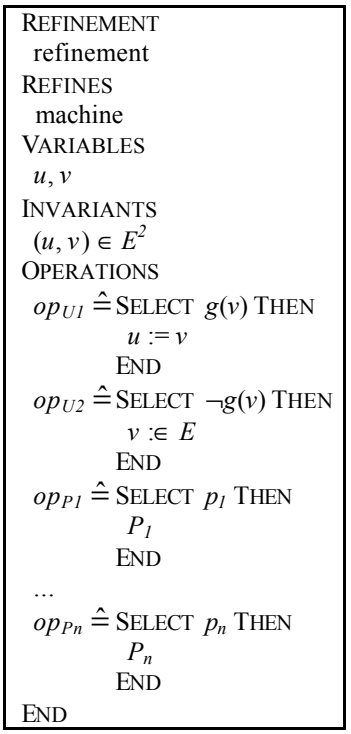
```
REFINEMENT
 refinement
REFINES
 machine
OPERATIONS
 op_S,T ≙          IF g THEN
          S
          ELSE
           T
          END
 op_P1 ≙ SELECT p_1 THEN
          P_1
          END
 ...
 op_Pn ≙ SELECT p_n THEN
          P_n
          END
END
```

4.6 Splitting by iteration

By applying theorem 6, we transform a single process formed of an unbounded choice into two processes that will, after some iterations of the abstract system, produce the same results.



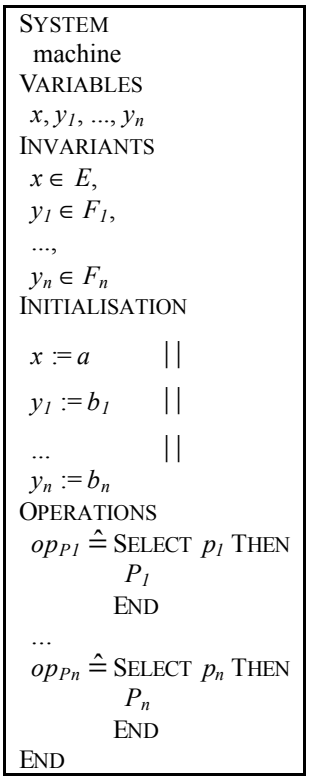
SYSTEM
machine
VARIABLES
u
INVARIANTS
$u \in E$
OPERATIONS
$op_U \mathrel{\hat{=}}$ ANY v WHERE
$g(v)$
THEN
$u := v$
END
$op_{P1} \mathrel{\hat{=}}$ SELECT p_1 THEN
P_1
END
...
$op_{Pn} \mathrel{\hat{=}}$ SELECT p_n THEN
P_n
END
END



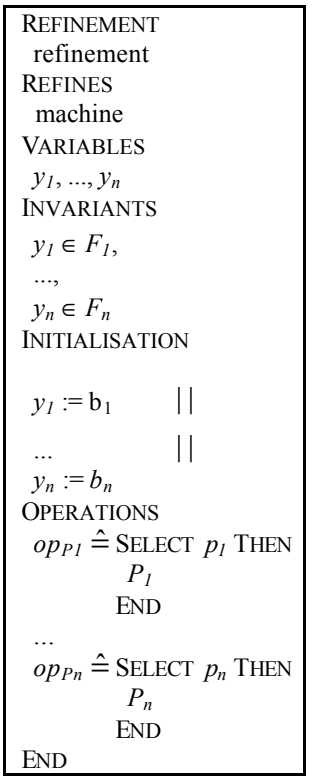
REFINEMENT
refinement
REFINES
machine
VARIABLES
u, v
INVARIANTS
$(u, v) \in E^2$
OPERATIONS
$op_{U1} \mathrel{\hat{=}}$ SELECT $g(v)$ THEN
$u := v$
END
$op_{U2} \mathrel{\hat{=}}$ SELECT $\neg g(v)$ THEN
$v :\in E$
END
$op_{P1} \mathrel{\hat{=}}$ SELECT p_1 THEN
P_1
END
...
$op_{Pn} \mathrel{\hat{=}}$ SELECT p_n THEN
P_n
END
END

4.7 Variables suppression

Variables defined in a abstract system can be no longer used in a more concrete one. We have to provide some means to automatically eliminate them. This can be done if the variable respects the following conditions: the initialisation of the variable validates just part or even all the guards of all the operations of the concrete system; the value of the variable is invariant within all the operations.



SYSTEM
machine
VARIABLES
$x, y_1, ..., y_n$
INVARIANTS
$x \in E,$
$y_1 \in F_1,$
...,
$y_n \in F_n$
INITIALISATION
$x := a$ ||
$y_1 := b_1$ ||
... ||
$y_n := b_n$
OPERATIONS
$op_{P1} \mathrel{\hat{=}}$ SELECT p_1 THEN
P_1
END
...
$op_{Pn} \mathrel{\hat{=}}$ SELECT p_n THEN
P_n
END
END



REFINEMENT
refinement
REFINES
machine
VARIABLES
$y_1, ..., y_n$
INVARIANTS
$y_1 \in F_1,$
...,
$y_n \in F_n$
INITIALISATION
$y_1 := b_1$ ||
... ||
$y_n := b_n$
OPERATIONS
$op_{P1} \mathrel{\hat{=}}$ SELECT p_1 THEN
P_1
END
...
$op_{Pn} \mathrel{\hat{=}}$ SELECT p_n THEN
P_n
END
END

Furthermore, if the variable validates the entire guards of all the operations, we have just attained the implementation level (called B_0). All the guards can be suppressed and replaced by scope operators (BEGIN, END).

5 Methodology

The automation of the previous operators can be quite hard to implement without prerequisites for all refinement rules. Nevertheless, we propose some guidelines for applying the distributed refinement.

1. Split the initial abstract system into a large number of guarded operations:
   a. Split unbounded choices in smaller ones (*splitting by selection*).
   b. Transform unbounded choices (*splitting by iteration*).
2. Try as much as possible to apply algorithmic and data refinements on those smaller events (use the Atelier B for the proofs).
3. From now on, group the processes:
   a. Grouping processes by loop is quite simple to use due to its prerequisites.
   b. Grouping processes by sequence is quite simple to use due to its prerequisites.
   c. Group processes by condition whenever their guards are complementary.
   d. Group processes by selection if their guards are complementary so you can group them by condition.
4. Create events whenever it is suitable:
   a. To add details from the specification.
   b. To adjust the grain of atomicity.

6 A case study: constructing correct ABR protocols

This case study describes the construction of an algorithm dedicated to the evaluation of the transfer rate of information in a network: namely the Average Bit Rate (ABR) control protocol. Our intention here is not to prove the process of development using the B Method (it has already been done in [13, 14]), but to illustrate how the distributed refinement is used for the automatic generation of a correct protocol. We won't prove each abstract system and algorithmic or data refinement, we will only highlight how distributed refinement simplifies the construction of distributed software. Note also that we multiply abstract systems and refinements in a pedagogical concern.

6.1 Average Bit Rate

The ABR algorithm is used to specify every node of an ATM network the transfer rate they must use from now on for all their communications. All the rates received by a node form some sort of a list defined as following:

$$\{(t_i, v_i) \mid t_i \geq 0 \wedge t_i \leq T\}$$

The bit rate v_i was received at the time t_i. La last value was received at a time lesser than or equal to the current time T.

For practical reasons, received values can not be processed immediately, but after a certain delay τ_3. The same way, received bit rates are meaningless after a certain delay τ_2. That is to say that only values received in the temporal window $](T-\tau_2), (T-\tau_3)]$ are meaningful.

Finally, the ABR algorithm gives only an approximation of the transfer rate in the form of a value A greater than or equal to the maximum of all the rates received within the temporal window.

We give extensively hereafter all refinements steps for applying the above defined operators in order to infer in a constructive way a correct algorithm for ABR.

6.2 Initial specification

We present first a relatively abstract model of the ABR algorithm. We formalize the definition of the previous section without taking time into account. It will be introduce later, in an ulterior step of refinement.

SYSTEM
 Abr1
VARIABLES
 l, v, j, k, A
INVARIANTS
 $l \in \mathrm{N}$
 $v \in (1..l) \rightarrow \mathrm{N}$
 $j \in 1..l$
 $k \in 1..l$
 $A \in \mathrm{N}$
 $\mathsf{max}(v[j..k]) \leq A$
INITIALISATION
 $j = k = l = 1$ ||
 $A = v(1)$
OPERATIONS
 ext $\widehat{=}$ Any X Where
  $X \in \mathrm{N}$
 Then
  $l, v(l+1) := l+1, X$
 End
 in $\widehat{=}$ Select $k < l$ Then
  $A :(\mathsf{max}(v[j..k+1]) \leq A)$ ||
  $k := k+1$
 End
 out $\widehat{=}$ Select $j < k$ Then
  $A :(\mathsf{max}(v[j+1..k]) \leq A$ ||
  $k := j+1$
 End
 inout $\widehat{=}$ Select $j < k \wedge k < l$ Then
  $A :(\mathsf{max}(v[j+1..k+1]) \leq A$ ||
  $j, k := j+1, k+1$
 End
End

6.3 First refinement

The first *distribution-oriented refinement* steps consist in simplifying or eliminating unbounded choices. For instance, we can refine the three operations in, out and inout, where $A : P(A)$ is syntactic sugar for a any construction on which we can apply refinement rule 6: *splitting by iteration*.

REFINEMENT
 Arb2 (iterating)
REFINES
 Abr1
VARIABLES
 l, v, j, k, A
 A'
INVARIANTS
 $l \in \mathrm{N}$
 $v \in (1..l) \rightarrow \mathrm{N}$
 $j \in 1..l$
 $k \in 1..l$
 $A \in \mathrm{N}$
 $A' \in \mathrm{N}$
 $\mathsf{max}(v[j..k]) \leq A$
OPERATIONS
 ext $\widehat{=}$ ANY X WHERE
  $X \in \mathrm{N}$
 THEN
  $l, v(l+1) := l+1, X$
 END
 in1 $\widehat{=}$ SELECT $k < l \wedge \mathsf{max}(v[j..k+1]) \leq A'$ THEN

```
        A := A' ||
        k := k + 1
      END
in2 ≙ SELECT k < l ∧ A' < max(v[j..k+1]) THEN
        A' :∈ N
      END
out1 ≙ SELECT j < k ∧ max(v[j+1..k]) ≤ A' THEN
        A := A' ||
        j := j + 1
      END
out2 ≙ SELECT j < k ∧ A' < max(v[j+1..k) THEN
        A' :∈ N
      END
inout1 ≙ SELECT j < k ∧ k < l ∧ max(v[j+1..k+1]) ≤ A' THEN
        A := A' ||
        j, k := j + 1, k + 1
      END
inout2 ≙ SELECT j < k ∧ k < l ∧ A' < max(v[j+1..k+1]) THEN
        A' :∈ N
      END
END
```

6.4 Second refinement

The six events in1, in2, out1, out2, inout1 and inout2 can now be simplified as A' can be eliminated.

- in1 reflects the arrival of a new value in the sensitive zone greater than the previous ones.
- out1 denotes an empty sensitive zone.
- inout1 is a mix of the previous cases.

We can, then, deduce the refinement of those events and obtain the following abstract machine.

```
REFINEMENT
  Abr3 (data refinement)
REFINES
  Abr2
VARIABLES
  l, v, j, k, A
INVARIANTS
  l ∈ N
  v ∈ (1..l) → N
  j ∈ 1..l
  k ∈ 1..l
  A ∈ N
  max(v[j..k]) ≤ A
OPERATIONS
  ext ≙ ANY X WHERE
        X ∈ N
      THEN
        l, v(l + 1) := l + 1, X
      END
  in1 ≙ SELECT k < l ∧ A < v(k+1) THEN
        A := v(k+1) ||
        k := k + 1
      END
  in2 ≙ SELECT k < l ∧ v(k+1) ≤ A THEN
        k := k + 1
      END
  out1 ≙ SELECT j < k ∧ j + 1 = l THEN
        A := v(l) ||
```

```
        j := j + 1
      END
 out2 ≙ SELECT j < k ∧ j + 1 ≠ l THEN
        j := j + 1
      END
 inout1 ≙ SELECT j < k ∧ k < l ∧ A < v(k + 1) THEN
        A := v(k + 1) ||
        j, k := j + 1, k + 1
      END
 inout2 ≙ SELECT j < k ∧ k < l ∧ v(k+1) ≤ A THEN
        j, k := j + 1, k + 1
      END
END
```

6.5 Third refinement

This is a data refinement. See [14] for explanations.

```
REFINEMENT
 Abr4 (data refinement)
REFINES
 Abr3
VARIABLES
 F, L, f, j, k, l, A
INITIALISATION
 F, L ∈ N x N
 f ∈ 1..l
 k < f ⇔ (k < l ∧ A < max(v[k+1..l])
 k < f ⇒ f = min( {n | n ∈ k + 1..l ∧ A < v(n) }) ∧ F = max(v[k+1..l])
 L = v(l)
OPERATIONS
 ext ≙ ANY X WHERE
        X ∈ N
       THEN
        l, L := l + 1, X ||
        CHOICE
          SELECT k < F ∧ F < X THEN
            F := X
          END
        OR
          SELECT f ≤ k ∧ A < X THEN
            f, F := l + 1, X
          END
        END
       END
 in1 ≙ SELECT k < l ∧ k + 1 = f THEN
        k, A := k + 1, F
       END
 in2 ≙ SELECT k < l ∧ k + 1 ≠ f THEN
        k := k + 1
       END
 out1 ≙ SELECT j < k ∧ j + 1 = l THEN
        j, A := j + 1, L
       END
 out2 ≙ SELECT j < k ∧ j + 1 ≠ l THEN
        j := j + 1
       END
 inout1 ≙ SELECT j < k < l ∧ k + 1 = f THEN
          j, k, A := j+1, k+1, F
        END
 inout2 ≙ SELECT j < k < l ∧ k + 1 ≠ f THEN
```

```
        j, k := j+1, k+1
      END
END
```

6.6 Fourth refinement

Here, we will try to simplify then event ext. To do so, we are going to apply rule of refinement 4: *splitting by selection*. We must, previously, transform the inner operations of the unbounded choice in the bounded choice of guarded operations. If is trivialy done by integrating the parallel affectation in each guarded operations and to create a third one whose guard is the complement of the two others. In fact, we create two events to get four disjunctive guards.

```
REFINEMENT
 Abr5 (selecting)
REFINES
 Abr4
VARIABLES
 F, L, f, j, k, l, A
INITIALISATION
 F, L ∈ N x N
 f ∈ 1..l
 k < f ⇔ (k < l ∧ A < max(v[k+1..l])
 k < f ⇒ f = min( {n | n ∈ k + 1..l ∧ A < v(n)}) ∧ F = max(v[k+1..l])
 L = v(l)
OPERATIONS
 ext1 ≙ ANY X WHERE
          X ∈ N ∧ k < f ∧ F < X
        THEN
          l, F, L := l+1, X, X
        END
 ext2 ≙ ANY X WHERE
          X ∈ N ∧ k < f ∧ X ≤ F
        THEN
          l, L := l+1, X
        END
 ext3 ≙ ANY X WHERE
          X ∈ N ∧ f ≤ k ∧ A < X
        THEN
          l, f, F, L := l+1, l+1, X, X
        END
 ext4 ≙ ANY X WHERE
          X ∈ N ∧ f ≤ k ∧ X ≤ A
        THEN
          l, L := l+1, X
        END
 in1 ≙ SELECT k < l ∧ k + 1 = f THEN
         k, A := k + 1, F
       END
 in2 ≙ SELECT k < l ∧ k + 1 ≠ f THEN
         k := k + 1
       END
 out1 ≙ SELECT j < k ∧ j + 1 = l THEN
          j, A := j + 1, L
        END
 out2 ≙ SELECT j < k ∧ j + 1 ≠ l THEN
          j := j + 1
        END
 inout1 ≙ SELECT j < k < l ∧ k + 1 = f THEN
            j, k, A := j+1, k+1, F
          END
 inout2 ≙ SELECT j < k < l ∧ k + 1 ≠ f THEN
```

```
        j, k := j+1, k+1
      END
END
```

6.7 Fifth refinement

Once again, a data refinement. See [14] for explanations.

```
REFINEMENT
  Abr6 (data refinement)
REFINES
  Abr5
VARIABLES
  T, t, l, j, k
INVARIANTS
  T ∈ N
  t ∈ (1..l) → N
  ∀ n · (n ∈ 1..l - 1 ⇒ t(n) < t(n+1))
  t(l) ≤ T
  k = max( {i | i ∈ 1..l ∧ t(i) + τ3 ≤ T} )
  j = max( {i | i ∈ 1..l ∧ t(i) + τ2 ≤ T} )
  τ3 < τ2
OPERATIONS
  ext1 ≙ ANY X WHERE
           X ∈ N ∧ T < t(f) + τ3 ∧ F < X ∧ t(l) ≠ T
         THEN
           l, F, L := l+1, X, X ||
           t(l+1) := T
         END
  ext2 ≙ ANY X WHERE
           X ∈ N ∧ T < t(f) + τ3 ∧ X ≤ F ∧ t(l) ≠ T
         THEN
           l, L := l+1, X ||
           t(l+1) := T
         END
  ext3 ≙ ANY X WHERE
           X ∈ N ∧ t(f) + τ3 ≤ T ∧ A < X ∧ t(l) ≠ T
         THEN
           l, f, F, L := l+1, l+1, X, X ||
           t(l+1) = T
         END
  ext4 ≙ ANY X WHERE
           X ∈ N ∧ t(f) + τ3 ≤ T ∧ X ≤ A ∧ t(l) ≠ T
         THEN
           l, L := l+1, X ||
           t(l+1) := T
         END
  in1 ≙ SELECT
          k < l ∧
          j < l ∧
          T + 1 = t(k+1) + τ3 ∧
          T + 1 ≠ t(j+1) + τ2 ∧
          T + 1 = t(f) + τ3 ∧
          T + 1 ≠ t(l) + τ2
        THEN
          k, A, T := k+1, F, T+1
        END
  in2 ≙ SELECT
          k < l ∧
```

```
        j < l ∧
        T + 1 = t(k+1) + τ3 ∧
        T + 1 ≠ t(j+1) + τ2 ∧
        T + 1 ≠ t(f) + τ3 ∧
        T + 1 ≠ t(l) + τ2
      THEN
        k, T := k+1, T+1
      END
out1 ≙ SELECT
        T + 1 = t(j+1) + τ2 ∧
        T + 1 = t(l) + τ2
      THEN
        j, A, T := j+1, L, T+1
      END
out2 ≙ SELECT
        j < l ∧
        k < l ⇒ T + 1 ≠ t(k+1) + τ3 ∧
        T + 1 = t(j+1) + τ2 ∧
        T + 1 ≠ t(f) + τ3 ∧
        T + 1 ≠ t(l) + τ2
      THEN
        j, T := j+1, T+1
      END
inout1 ≙ SELECT
          k < l ∧
          j < l ∧
          T + 1 = t(k+1) + τ3 ∧
          T + 1 = t(j+1) + τ2 ∧
          T + 1 = t(f) + τ3 ∧
          T + 1 ≠ t(l) + τ2
        THEN
          j, k, A, T := j+1, k+1, F, T+1
        END
inout2 ≙ SELECT
          k < l ∧
          j < l ∧
          T + 1 = t(k+1) + τ3 ∧
          T + 1 = t(j+1) + τ2 ∧
          T + 1 ≠ t(f) + τ3 ∧
          T + 1 ≠ t(l) + τ2
        THEN
          j, k, T := j+1, k+1, T+1
        END
tick ≙ SELECT
        k < l ⇒ T + 1 ≠ t(k+1) + τ3 ∧
        j < l ⇒ T + 1 ≠ t(j+1) + τ2 ∧
        T + 1 ≠ t(f) + τ3 ∧
        T + 1 ≠ t(l) + τ2
      THEN
        T := T + 1
      END
END
```

6.8 Sixth refinement

We will now group events together:

- The guards of in2, ou2, inout2 and tick can be factorized.
- The guards of in1 and inout1 can also be factorized.

```
REFINEMENT
 Abr7 (conditioning)
REFINES
 Abr6
OPERATIONS
 ext1 ≙ ANY X WHERE
          X ∈ N ∧ T < t(f) + τ3 ∧ F < X ∧ t(l) ≠ T
         THEN
          l, F, L := l+1, X, X ||
          t(l+1) := T
         END
 ext2 ≙ ANY X WHERE
          X ∈ N ∧ T < t(f) + τ3 ∧ X ≤ F ∧ t(l) ≠ T
         THEN
          l, L := l+1, X ||
          t(l+1) := T
         END
 ext3 ≙ ANY X WHERE
          X ∈ N ∧ t(f) + τ3 ≤ T ∧ A < X ∧ t(l) ≠ T
         THEN
          l, f, F, L := l+1, l+1, X, X ||
          t(l+1) = T
         END
 ext4 ≙ ANY X WHERE
          X ∈ N ∧ t(f) + τ3 ≤ T ∧ X ≤ A ∧ t(l) ≠ T
         THEN
          l, L := l+1, X ||
          t(l+1) := T
         END
 in2_out2_inout2_tick ≙ SELECT T + 1 ≠ t(f) + τ3 ∧ T + 1 ≠ t(l) + τ2 THEN
                          T := T + 1 ||
                          CHOICE
                           k := k + 1
                          OR
                           j := j + 1
                          OR
                           j, k := j+1, k+1
                          OR
                           skip
                          END
                         END
 in1_inout1 ≙ SELECT T + 1 ≠ t(f) + τ3 THEN
                k, A, T := k+1, F, T+1 ||
                CHOICE
                 j := j + 1
                OR
                 skip
                END
               END
 out1 ≙ SELECT
          T + 1 = t(j+1) + τ2 ∧
          T + 1 = t(l) + τ2
         THEN
          j, A, T := j+1, L, T+1
         END
END
```

6.9 Seventh refinement

The last data refinement. See [14] for explanations.

```
REFINEMENT
  Abr8 (data refinement)
REFINES
  Abr7
VARIABLES
  TF, TL, T, F, L, τ2, τ3, A
OPERATIONS
  ext1 ≙ ANY X WHERE
           X ∈ N ∧ T < TF ∧ F < X ∧ TL ≠ T + τ2
         THEN
           F, L, TL := X, X, T + τ2
         END
  ext2 ≙ ANY X WHERE
           X ∈ N ∧ T < TF ∧ X ≤ F ∧ TL ≠ T + τ2
         THEN
           L, TL := X, T + τ2
         END
  ext3 ≙ ANY X WHERE
           X ∈ N ∧ TF ≤ T ∧ A < X ∧ TL ≠ T + τ2
         THEN
           F, L, TL, TF := X, X, T + τ2, T + τ3
         END
  ext4 ≙ ANY X WHERE
           X ∈ N ∧ TF ≤ T ∧ X ≤ A ∧ TL ≠ T + τ2
         THEN
           L, TL := X, T + τ2
         END
  in2_out2_inout2_tick ≙ SELECT T + 1 ≠ TF ∧ T + 1 ≠ TL THEN
                           T := T + 1
                         END
  in1_inout1 ≙ SELECT T + 1 = TF THEN
                 A, T := F, T + 1
               END
  out1 ≙ SELECT T + 1 = TL THEN
           A, T := L, T + 1
         END
END
```

6.10 Last refinement: implementation

We group once more. in2_out2_inout2_tick, in1_inout1 and out1 becomes the scheduler. The four ext events group into the simplified ABR algorithm.

```
REFINEMENT
  Abr9
REFINES
  Abr8
VARIABLES
  A, T, F, X, TF, TL, τ2, τ3
INVARIANTS
  TF = t(f) + τ3
  TL = t(l) + τ2
OPERATIONS
  scheduler ≙ BEGIN
                T := T + 1 ||
                IF T + 1 = TF THEN
                  A := F
                ELSE
                  IF T + 1 = TL THEN
                    A := L
```

```
                END
              END
            END
abr ≙ ANY X WHERE
        X ∈ N ∧ TL ≠ T + τ2
      THEN
        L, TL := X, T + τ2 ||
        IF T < TF THEN
          IF F < X THEN
            F := X
          END
          ELSE
            IF A < X THEN
              F, TF := X, T + τ3
            END
          END
        END
END
```

7 Conclusion

We have presented a new refinement operator dedicated for the derivation of distributed / multi-process / multi-threaded software from their specifications. The methodological approach we are adopting follows some essential choices.

First, we do not build one abstract machine for each component of a distributed system and prove afterwards the correction of their interactions. The components will be generated progressively in the successive refinements. That way, "communications" between processes are correct by construction.

Second, we intensively use the "divide and conquer" paradigm; we split as much as possible specifications into very simple events. That way, algorithmic and data refinements are easier to prove; preservation of invariants is easier to ensure. It is only after that we group events to obtain standard programming structures: loops, conditional statements and sequences.

Third, communications are taken into account in a very simple manner, through global variables. Of course, one can deal with synchronization and coordination, but data exchanges are still quite limited. Introduction of more sophisticated mechanisms, such as monitors, channels of communication [1, 2, 7], remote procedure call and remote method invocation, is to be considered.

Furthermore, our memory is shared, which is not very realistic. We plan to implement proceedings to manage with distributed memory.

This paper foreshadows an overall method to specify and construct critical distributed systems.

All the previous mechanisms do not allow specifying and developing *time critical* systems. We consider also the introduction of time in our specifications by means of delays [12].

There is not yet the slightest notion of localization of the different processes. We should be able to split a whole abstract system in several others where each would represent a node in our topology.

References

[1] A. W. Roscoe. *The Theory and Practice of Concurrency*. International Series in Computer Science. Prentice-Hall, 1997.

[2] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[3] J.-R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[4] J.-R. Abrial. Extending B without changing it (for developing Distributed Systems). In Henri Abrias, editor, *Proceedings of the 1st Conference on the B Method*, Putting into practice methods and tools for information system design, pages 169-191, 3 rue du Maréchal Joffre, BP 34103, 44041 Nantes Cedex 1, November 1996. IRIN Institut de Recherche en Informatique de Nantes.

[5] J.-R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In Didier Bert, editor, *B'98 : The 2nd International B Conference*, Recent Advances in the Development and Use of the B Method, pages 83 – 128, Montpellier, April 1998. LIRRM Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier.

[6] K. Lano, J. Fiadeiro and J. Dick. Extending B AMN with concurrency. Dept. of Computing, Imperial College, 1996.

[7] M. Butler. csp2B: A Practical Approach To Combining CSP and B. In J. Wing, J. Woodcock and J. Davies, editors, *FM'99 World Congress on Formal Methods*, pages 223 – 241, September 1999.

[8] M. Butler and M. Walden. Distributed System Development in B. In Henri Abrias, editor, *Proceedings of the 1st Conference on the B Method*, Putting into practice methods and tools for information system design, pages 169-191, 3 rue du Maréchal Joffre, BP 34103, 44041 Nantes Cedex 1, November 1996. IRIN Institut de Recherche en Informatique de Nantes.

[9] P. Behm, P. Benoit, A. Faivre and J.-M. Meynadier. METEOR: A successful application of B in a large p roject. In *Proceedings of FM'99: World Congress on Formal Methods*, pages 369 – 387, 1999.

[10] R. J. R. Back and J. von Wright. Refinement Calculus, part I: Sequential nondeterministic programs. In J. W. de Bakker, W.-P. de Roever and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 42 – 66. springer-Verlag, 1990.

[11] R. J. R. Back and J. von Wright. Refinement Calculus, part II: Parallel and Reactive Programs. In J. W. de Bakker, W.-P. de Roever and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 67 – 93. springer-Verlag, 1990.

[12] T. Bolognesi. Timed LOTOS: Which way to go. In *Workshop on LOTOS*, London, UK, 1990. British Computer Society.

[13] J-R. Abrial. *Développement de l'algorithme ABR (forme élaborée)*, 1999.

[14] J-R. Abrial. *Développement de l'algorithme ABR (forme simplifiée)*, 1999.

Appendix: Proofs

Proof of theorem 1

Lemma 1. Let P be a process. The opening of the sequence of P and the opening of P is equal to the sequence of the opening of P and the opening of the sequence of P and the opening of P.

$(P ; P^{\wedge})^{\wedge} = P^{\wedge} ; (P ; P^{\wedge})^{\wedge}$

Proof.

$P^{\wedge} ; (P ; P^{\wedge})^{\wedge}$

$=$ unfolding property

$(\mathsf{skip} \Box (P ; P^{\wedge})) ; (P ; P^{\wedge})^{\wedge}$

$=$ distributivity

$(P ; P^{\wedge})^{\wedge} \Box ((P ; P^{\wedge}) ; (P ; P^{\wedge})^{\wedge})$

$=$ unfolding property

$\mathsf{skip} \Box ((P ; P^{\wedge}) ; (P ; P^{\wedge})^{\wedge}) \Box ((P ; P^{\wedge}) ; (P ; P^{\wedge})^{\wedge})$

$=$ $P \Box P = P$ and associativity

$\mathsf{skip} \Box ((P ; P^{\wedge}) ; (P ; P^{\wedge})^{\wedge})$

$=$ unfolding property

$0(P ; P^{\wedge})^{\wedge}$

□

Lemma 2. Let P be a process. The opening of P is equal to the opening of the sequence of P and the opening of P.

$P^{\wedge} = (P ; P^{\wedge})^{\wedge}$

Proof.

$(P ; P^{\wedge})^{\wedge} \Box P^{\wedge}$

$\Leftarrow$ monotonicity

$1P ; P^{\wedge} \Box P$

$\Leftarrow$ monotonicity and neutral element

$P^{\wedge} \Box \mathsf{skip}$

$\Leftrightarrow$ unfolding property

$(P ; P^{\wedge}) \Box \mathsf{skip} \Box \mathsf{skip}$

$\Leftrightarrow$ definition 3, $\forall\, q \subseteq s$

$(P ; P^{\wedge})(q) \text{Ç } q \text{ Í } q$

$\Leftrightarrow$ set theory

true

$(P ; P^{\wedge})^{\wedge} \Box P^{\wedge}$ (a)

$P^\hat{} \sqsubseteq (P ; P^\hat{})^\hat{}$

$\Leftrightarrow$ unfolding property

$P^\hat{} \sqsubseteq P ; P^\hat{} ; (P ; P^\hat{})^\hat{} \sqcap \mathsf{skip}$

$\Leftrightarrow$ lemma 1

$P^\hat{} \sqsubseteq P ; (P ; P^\hat{})^\hat{} \sqcap \mathsf{skip}$

$\Leftrightarrow$ definitions of $\sqsubseteq$ and $\hat{}$, $\forall\, q \subseteq s$

$\mathsf{fix}(q \mid P) \subseteq (P ; (P ; P^\hat{})^\hat{} \sqcap \mathsf{skip})(q)$

$\Leftarrow$ fix point property

$(q \mid P)((P ; (P ; P^\hat{})^\hat{} \sqcap \mathsf{skip})(q)) \subseteq (P ; (P ; P^\hat{})^\hat{} \sqcap \mathsf{skip})(q)$

$\Leftrightarrow$ definitions of $\mid$, $\sqcap$ and skip

$q \cap P((P ; (P ; P^\hat{})^\hat{} \sqcap \mathsf{skip})(q)) \subseteq (P ; (P ; P^\hat{})^\hat{})(q) \cap q$

$\Leftarrow$ set theory

$P((P ; (P ; P^\hat{})^\hat{} \sqcap \mathsf{skip})(q)) \subseteq P((P ; P^\hat{})^\hat{}(q))$

$\Leftrightarrow$ unfolding property

$P((P ; (P ; P^\hat{})^\hat{} \sqcap \mathsf{skip})(q)) \subseteq P((P ; P^\hat{} ; (P ; P^\hat{})^\hat{} \sqcap \mathsf{skip})(q))$

$\Leftrightarrow$ lemma 1

$P((P ; (P ; P^\hat{})^\hat{} \sqcap \mathsf{skip})(q)) \subseteq P((P ; (P ; P^\hat{})^\hat{} \sqcap \mathsf{skip})(q))$

$\Leftrightarrow$ set theory

true

$P^\hat{} \sqsubseteq (P ; P^\hat{})^\hat{}$ (b)

$P^\hat{} = (P ; P^\hat{})^\hat{}$

$\Leftrightarrow$ $\sqsubseteq$ property

$2(P ; P^\hat{})^\hat{} \sqsubseteq P^\hat{} \sqcup P^\hat{} \sqsubseteq (P ; P^\hat{})^\hat{}$

$\Leftrightarrow$ (a) and (b)

true

□

Lemma 3. Let P and Q be two processes. The opening of the choice of P and Q is refined by the opening of the sequence of P and the opening of Q.

$(P \sqsubseteq Q)^\hat{} \sqsubseteq (P ; Q^\hat{})^\hat{}$

Proof.

$(P \sqsubseteq Q)^\hat{} \sqsubseteq (P ; Q^\hat{})^\hat{}$

$\Leftrightarrow$ lemma 2

$3((P \sqsubseteq Q) ; (P \sqsubseteq Q)^\hat{})^\hat{} \sqsubseteq (P ; Q^\hat{})^\hat{}$

$\Leftarrow$ monotonicity

4$(P \sqsubset Q) ; (P \sqsubset Q)^{\wedge} \sqsubset P ; Q^{\wedge}$

$\Leftarrow$ monotonicity

5$(P \sqsubset Q \sqsubset P) \dot{\cup} (P \sqsubset Q)^{\wedge} \sqsubset Q^{\wedge}$

$\Leftarrow$ monotonicity

$(P \sqsubset Q \sqsubset P) \dot{\cup} (P \sqsubset Q \sqsubset Q)$

$\Leftrightarrow$ $\sqsubset$property

true

□

Lemma 4. Let P, Q and R be three processes. The opening of the choice of P, Q and R is refined by the opening of the choice of P and the sequence of Q and the opening of R.

$(P \sqsubset Q \sqsubset R)^{\wedge} \sqsubset (P \sqsubset (Q ; R^{\wedge}))^{\wedge}$

Proof.

$(P \sqsubset Q \sqsubset R)^{\wedge} \sqsubset (P \sqsubset (Q ; R^{\wedge}))^{\wedge}$

$\Leftarrow$ lemma 3

6$((P \sqsubset Q) ; R^{\wedge})^{\wedge} \sqsubset (P \sqsubset (Q ; R^{\wedge}))^{\wedge}$

$\Leftarrow$ monotonicity

$(P \sqsubset Q) ; R^{\wedge} \sqsubset P \sqsubset (Q ; R^{\wedge})$

$\Leftrightarrow$ distributivity

$(P ; R^{\wedge}) \sqsubset (Q ; R^{\wedge}) \sqsubset P \sqsubset (Q ; R^{\wedge})$

$\Leftarrow$ monotonicity

$(P ; R^{\wedge} \sqsubset P) \dot{\cup} (Q ; R^{\wedge} \sqsubset Q ; R^{\wedge})$

$\Leftrightarrow$ definition 3, $\forall\, q \subseteq$ s and definition of ;

0$P(R^{\wedge}(q)) \subseteq P(q) \wedge \mathsf{true}$

$\Leftrightarrow$ monotonicity

$R^{\wedge}(q)$ Í q

$\Leftrightarrow$ definition of $^{\wedge}$

$\mathsf{fix}(q \mid R) \subseteq q$

$\Leftrightarrow$ fix point property and definition of |

$q \cap R(\mathsf{fix}(q \mid R)) \subseteq q$

$\Leftrightarrow$ set theory

true

□

Proof of theorem 1.

$P \sqsubset Q \overset{\wedge}{\preceq} P \sqsubset Q \sqsubset R$

$\Leftrightarrow$ definition of $\overset{\wedge}{\preceq}$ and monotonicity

7$(P \sqsubset Q)^{\wedge} \sqsubset (P \sqsubset Q \sqsubset R)^{\wedge}$

$\Leftrightarrow$ $\mathsf{skip}^{\wedge} = \mathsf{skip}$ and $Q ; \mathsf{skip} = Q$

$(P \sqsubset Q ; \mathsf{skip}^{\wedge})^{\wedge} \sqsubset (P \sqsubset Q \sqsubset R)^{\wedge}$

$\Leftarrow$ lemma 4

$(P \sqsubset Q \sqsubset \mathsf{skip})^{\wedge} \sqsubset (P \sqsubset Q \sqsubset R)^{\wedge}$

$\Leftarrow$ monotonicity

skip □ *R*

□

Proof of theorem 2

Lemma 5. Let *P* be a process. The opening of *P* is refined by *P*.

$P^{\wedge} \sqsubseteq P$

Proof.

$P^{\wedge} \sqsubseteq P$

⇔ unfolding property

$P ; P^{\wedge} \sqsubseteq \mathsf{skip} \sqsubseteq P$

⇐ $P \sqsubseteq \mathsf{skip} \sqsubseteq P$

$P ; P^{\wedge} \sqsubseteq \mathsf{skip} \sqsubseteq P \sqsubseteq \mathsf{skip}$

⇐ monotonicity

8 $P ; P^{\wedge} \sqsubseteq P$

⇐ monotonicity and neutral element

$P^{\wedge} \sqsubseteq \mathsf{skip}$

⇔ unfolding property

$P ; P^{\wedge} \sqsubseteq \mathsf{skip} \sqsubseteq \mathsf{skip}$

⇔ property of □

true

□

Lemma 6. Let *P*, *Q* and *R* be three processes. The bounded choice of *P*, *Q* and *R* is distributed refined by the bounded choice of *P* and a process formed of the sequence of *Q* and *R*.

$P \sqsubseteq Q \sqsubseteq R \mathrel{\hat{\preceq}} P \sqsubseteq (Q ; R)$

Proof.

$P \sqsubseteq Q \sqsubseteq R \mathrel{\hat{\preceq}} P \sqsubseteq Q ; R$

⇐ monotonicity

$(P \sqsubseteq Q \sqsubseteq R)^{\wedge} \sqsubseteq ((P ; Q) \sqsubseteq R)^{\wedge}$

⇐ lemma 4

$((P ; Q^{\wedge}) \sqsubseteq R)^{\wedge} \sqsubseteq ((P ; Q) \sqsubseteq R)^{\wedge}$

⇐ monotonicity

9 $Q^{\wedge} \sqsubseteq Q$

⇔ lemma 5

true

□

Proof of theorem 2.

$P \sqsubseteq (g \Rightarrow Q) \sqsubseteq (h \Rightarrow R) \mathrel{\hat{\preceq}} P \sqsubseteq (g \Rightarrow (Q ; R))$

⇐ lemma 6

$P \Box ((g \triangleright Q) ; (h \triangleright R)) \mathrel{\hat{\preceq}} P \Box (g \triangleright (Q ; R))$

⇐ monotonicity

$P \Box ((g \triangleright Q) ; (h \triangleright R)) \sqsubseteq P \Box (g \triangleright (Q ; R))$

⇔ definition 3, $\forall q \subseteq s$

$P(q) \cap (\overline{g} \cup Q(\overline{h} \cup R(q))) \cap P(q) \cap (\overline{g} \cup Q(R(q)))$

⇔ if $\forall k \cdot (k \subseteq s \wedge Q(k) \subseteq Q(h))$

$P(q) \cap (\overline{g} \cup (Q(\overline{h} \cup R(k)) \cap Q(h))) \cap P(q) \cap \overline{g} \cup Q(R(q)))$

⇔ conjunctivity and set theory

$P(q) \cap (\overline{g} \cup Q(h \cap R(k))) \cap P(q) \cap (\overline{g} \cup (R(q)))$

⇔ conjunctivity

$P(q) \cap (\overline{g} \cup (Q(h) \cap Q(R(k)))) \cap P(q) \cap (\overline{g} \cup Q(R(q)))$

⇔ if $\forall k \cdot (k \subseteq s \wedge Q(k) \subseteq Q(h))$

$P(q) \cap (\overline{g} \cup Q(R(k))) \cap P(q) \cap (\overline{g} \cup Q(R(q)))$

⇔ set theory

true

□

Proof of theorem 3

Lemma 7. Let P be a process and g and h two predicates. The process formed by P guarded by the conjunction of g and h is equal to the process formed by P guarded by h, the whole guarded by g.

$((g \cup h) \triangleright P) = (g \triangleright (h \triangleright P))$

Proof.

$(g \wedge h) \Rightarrow P = g \Rightarrow (h \Rightarrow P)$

⇔ definition 3, $\forall q \subseteq s$

$((g \cup h) \triangleright P)(q) = (g \triangleright (h \triangleright P))(q)$

⇔ definition of $\Rightarrow$

$((g \cup h) \triangleright P)(q) = \overline{g} \cup \overline{h} \cup P(q)$

⇔ de Morgan

$((g \cup h) \triangleright P)(q) = \overline{g \cap h} \cup P(q)$

⇔ definition of $\Rightarrow$

$((g \cup h) \triangleright P)(q) = ((g \cup h) \triangleright P)(q)$

⇔

true

□

Lemma 8. Let P be a process and g a predicate. The opening of P, guarded by g, if g is invariant by P, refines the opening of P guarded by g.

$g \cap P(g) \triangleright (g \triangleright P)^{\wedge} \sqsubseteq g \triangleright P^{\wedge}$

Proof.

$(g \Rightarrow P)^{\wedge} \sqsubseteq g \Rightarrow P^{\wedge}$

⇔ definition 3, $\forall q \subseteq s$

$(g \triangleright P)^{\wedge}(q) \cap (g \triangleright P^{\wedge})(q)$

⇔ definitions of $^{\wedge}$ and $\Rightarrow$

$\mathsf{fix}(q \mid (g \Rightarrow P)) \subseteq \overline{g} \cup P^{\wedge}(q)$

⇐ fix point property

$(q \mid (g \triangleright P))(\overline{g} \cup P^{\wedge}(q)) \cap \overline{g} \cup P^{\wedge}(q)$

⇔ definitions of $\mid$ and $\Rightarrow$

$q \cap (\overline{g} \cup P(\overline{g} \cup P^{\wedge}(q))) \cap \overline{g} \cup P^{\wedge}(q)$

⇔ set theory

$q \cap g \cap P(\overline{g} \cup P^{\wedge}(q)) \subseteq g \cap P^{\wedge}(q)$

$\Leftrightarrow$ if $g \subseteq P(g)$

$q \cap g \cap P(g) \cap P(\overline{g} \cup P^{\wedge}(q)) \subseteq g \cap P^{\wedge}(q)$

$\Leftrightarrow$ conjunctivity and set theory

$q \cap g \cap P(g \cap P^{\wedge}(q)) \subseteq g \cap P^{\wedge}(q)$

$\Leftrightarrow$ conjunctivity

$q \cap g \cap P(g) \cap P(P^{\wedge}(q)) \subseteq g \cap P^{\wedge}(q)$

$\Leftrightarrow$ if $g \subseteq P(g)$

$q \cap g \cap P(P^{\wedge}(q)) \subseteq g \cap P^{\wedge}(q)$

$\Leftrightarrow$ folding property

$g \cap P^{\wedge}(q) \subseteq g \cap P^{\wedge}(q)$

$\Leftrightarrow$

true

□

Lemma 9. Let P and Q be two processes and g a predicate. The sequence of P and Q guarded by g is equal to the sequence of P and Q if P validates g.

$\langle\langle k \rangle\rangle (k \subseteq s \wedge P(k) \subseteq P(g)) \Rightarrow P ; (g \Rightarrow Q) = P ; Q$

Proof.

$P ; (g \Rightarrow Q) = P ; Q$

$\Leftrightarrow$ definition 3, $\forall q \subseteq s$

$(P ; (g \Rightarrow Q))(q) = (P ; Q)(q)$

$\Leftrightarrow$ definition of ; and $\Rightarrow$

$P(\overline{g} \cup Q(q)) = P(Q(q))$

$\Leftrightarrow$ if $\forall k \cdot (k \subseteq s \wedge P(k) \subseteq P(g))$

$P(g) \cap P(\overline{g} \cup Q(q)) = P(Q(q))$

$\Leftrightarrow$ conjunctivity and set theory

$P(g \cap Q(q)) = P(Q(q))$

$\Leftrightarrow$ conjunctivity

$P(g) \cap P(Q(q)) = P(Q(q))$

$\Leftrightarrow$ if $\forall k \cdot (k \subseteq s \wedge P(k) \subseteq P(g))$

$P(Q(q)) = P(Q(q))$

$\Leftrightarrow$

true

□

Lemma 10. Let P be a process and g a predicate. P is refined by the sequence of P and a process formed of skip guarded by g.

***P* ⊑ *P* ; (*g* ⇒ skip)**

Proof.

$P \sqsubseteq P ; (g \Rightarrow \mathsf{skip})$

$\Leftrightarrow$ definition 3, $\forall q \subseteq s$

$P(q) \subseteq (P ; (g \Rightarrow \mathsf{skip}))(q)$

$\Leftrightarrow$ definitions of ; and $\Rightarrow$

$P(q) \subseteq P(\overline{g} \cup \mathsf{skip}(q))$

$\Leftrightarrow$ definition of skip

$P(q) \subseteq P(\overline{g} \cup q)$

$\Leftarrow$ monotonicity

$q \subseteq \overline{g} \cup q$

$\Leftrightarrow$ set theory

true

□

Proof of theorem 3.

$(P \square Q \square (g \wedge h \Rightarrow R))^{\wedge}$

□ lemma 4

(P□ (Q ; (g Ù h Þ R)^))^

= lemma 7

(P□ (Q ; (g Þ (h Þ R))^))^

□ lemma 8 if $g \subseteq R(g)$

(P□ (Q ; (g Þ (h Þ R)^)))^

□ lemma 9 if $\forall k \cdot (k \subseteq s \wedge Q(k) \subseteq Q(g)$

(P□ (Q ; (h Þ R)^))^

□ lemma 10

$0(P \square (Q ; (h \Rightarrow R)^{\wedge} ; (\neg h \Rightarrow \text{skip})))^{\wedge}$

□

Proof of theorem 4

$P \square @x \cdot (x \in E \wedge g(x) \wedge h \Rightarrow S_x) \square @x \cdot (x \in E \wedge g(x) \wedge i \Rightarrow T_x)$

=

$P \square @x \cdot (x \in E \wedge g(x) \Rightarrow (h \Rightarrow S_x \square i \Rightarrow T_x))$

$\Leftrightarrow$ definition 3, $\forall q \subseteq s$

$$P(q) \cap \bigcap_{\{x \mid x \in E\}} (\overline{g(x) \cap h} \cup S_x(q)) \cap \bigcap_{\{x \mid x \in E\}} (\overline{g(x) \cap i} \cup T_x(q))$$

=

$$P(q) \cap \bigcap_{\{x \mid x \in E\}} (\overline{g(x)} \cup ((\overline{h} \cup S_x(q)) \cap (\overline{i} \cup T_x(q))))$$

$\Leftrightarrow$ set theory

$$P(q) \cap \bigcap_{\{x \mid x \in E\}} (\overline{g(x)} \cup ((\overline{h} \cup S_x(q)) \cap (\overline{i} \cup T_x(q))))$$

=

$$P(q) \cap \bigcap_{\{x \mid x \in E\}} (\overline{g(x)} \cup ((\overline{h} \cup S_x(q)) \cap (\overline{i} \cup T_x(q))))$$

$\Leftrightarrow$

true

□

Proof of theorem 5

$(P \Box (g \triangleright Q) \Box (\varnothing g \triangleright R))^{\wedge} = (P \Box ((g \triangleright Q) \Box (\varnothing g \triangleright Q)))^{\wedge}$

$\Leftrightarrow$ monotonicity

$P \Box (g \triangleright Q) \Box (\varnothing g \triangleright R) = P \Box ((g \triangleright Q) \Box (\varnothing g \triangleright Q))$

$\Leftrightarrow$ associativity

true

□

Proof of theorem 6

Lemma 11. Let u be a variable in set E. Let $g(x)$ be a predicate dependent on its parameter x. Let P and Q be two processes such that Q is the affectation to u of any value x verifying $g(x)$. The bounded choice of P and Q is distributed refined by the bounding choice of P and a process which, if $g(u)$ does not hold, affects to u a value of E.

$$P \Box u{:}g(u) \;\sqsubseteq\; P \Box (\neg g(u) \Rightarrow u :\in E)$$

Proof.

$P \Box u{:}g(u)$

$\sqsubseteq$

$\boldsymbol{P \Box \varnothing g(u) \triangleright u :\in E}$

$\Leftrightarrow$ definition of : and :∈

$P \Box @v \cdot (v \in E \wedge g(v) \triangleright u := v)$

$\sqsubseteq$

$P \Box \varnothing g(u) \triangleright @v \cdot (v \in E \triangleright u := v)$

$\Leftrightarrow$ monotonicity

$(P \Box @v \cdot (v \in E \wedge g(v) \Rightarrow u := v))^{\wedge}$

□

$(P \Box \varnothing g(u) \triangleright @v \cdot (v \in E \triangleright u := v))^{\wedge}$

$\Leftrightarrow$ definition 3, $\forall q \subseteq$ s and definition of $^{\wedge}$

$\mathsf{fix}\,(q \mid P \Box @v \cdot (v \in E \wedge g(v) \Rightarrow u := v))$

$\subseteq$

$(P \Box \varnothing g(u) \triangleright @v \cdot (v \in E \triangleright u := v))^{\wedge}(q)$

$\Leftarrow$ fix point property

$I = (P \Box \neg g(u) \Rightarrow @v \cdot (v \in E \Rightarrow u := v))^{\wedge}$

$(q \mid P \Box @v \cdot (v \in E \wedge g(v) \Rightarrow u{:=}\, v))(\mathrm{I}(q))$

$\subseteq$

$(P \Box \varnothing g(u) \triangleright @v \cdot (v \in E \triangleright u := v))^{\wedge}(q)$

$\Leftrightarrow$ unfolding property

$(q \mid P \Box @v \cdot (v \in E \wedge g(v) \Rightarrow u{:=}\, v))(I(q))$

$\subseteq$

$(((P \Box \neg g(u) \Rightarrow @v \cdot (v \in E \Rightarrow u := v)) \,;\, I) \Box \mathsf{skip})(q)$

⇔ definitions of |, ; and □

$q \cap P(I(q)) \cap (@v \cdot (v \in E \wedge g(v) \Rightarrow u := v))(I(q))$

⊆

q Ç $P(I(q))$ Ç $(g(u)$ È $(@v$ ᵡ $(v$ Î E Þ $u := v))(I(q)))$

⇐ set theory

$(@v \cdot (v \in E \wedge g(v) \Rightarrow u := v))(I(q))$

⊆

$(@v$ ᵡ $(v$ Î E Þ $u := v))(I(q))$

⇔ definition of @

$$\bigcap_{\{v \mid v \in E \wedge g(v)\}} (u := v)(I(q))$$

⊆

$$\bigcap_{\{v \mid v \in E\}} (u := v)(I(q))$$

⇔ set theory

true

□

Lemma 12. Let g be a predicate. skip is refined by skip guarded by the predicate g.

skip □ $g \Rightarrow$ skip

Proof.

skip □ $g \Rightarrow$ skip

⇔ by definition, $\forall q \subseteq$ s

skip$(q) \subseteq (g \Rightarrow$ skip$)(q)$

⇔ definition of $\Rightarrow$ and skip

q Í $\overline{g}$ È q

⇔ set theory

true

□

Lemma 13. Let u and v be two variables in E. Let $g(x)$ be a predicate dependent on its parameter x. Let P, Q and R be three processes such that:

- Q is the affectation to u of any value in E if $g(u)$ does not hold
- R does nothing (skip) if $g(u)$ holds.

The bounded choice of P, Q and R is distributed refined by the bounded choice of P and two processes S and T such that:

- S is the affectation to v of any value in E if $g(v)$ does not hold
- T is the affectation of v to u if $g(v)$ holds.

$$\overset{\wedge}{\preceq} \begin{array}{l} P \,□\, \neg g(u) \Rightarrow u :\in E \,□\, g(u) \Rightarrow \text{skip} \\ P \,□\, \neg g(v) \Rightarrow v :\in E \,□\, g(v) \Rightarrow u := v \end{array}$$

Proof.

$\neg g(u) \Rightarrow u :\in E \preceq \neg g(v) \Rightarrow v :\in E$

⇔ definition of $\preceq$

PR$(\neg g(u) \Rightarrow u :\in E)$ □ PR$(\neg g(v) \Rightarrow v :\in E)$

⇔ definition of PR

$@u$ ᵡ $(Øg(u)$ Þ $u :$ Î $E)$ □ $@v$ ᵡ $(Øg(v)$ Þ $v :$ Î $E)$

⇔ definition of @, $\forall q \subseteq$ s

$$\bigcap_{u \mid u \in E} (Øg(u) \text{ Þ } u :\text{Î } E)(q) \text{ Í } \bigcap_{v \mid v \in E} (Øg(v) \text{ Þ } v :\text{Î } E)(q)$$

⇔ set theory

true

$\neg g(u) \Rightarrow u :\in E \preceq \neg g(v) \Rightarrow v :\in E$ (a)

$g(u) \Rightarrow \mathsf{skip} \preceq g(v) \Rightarrow u := v$

$\Leftrightarrow$ definition of $\preceq$

$\mathsf{PR}(g(u) \Rightarrow \mathsf{skip}) \sqsubseteq \mathsf{PR}(g(v) \Rightarrow u := v)$

$\Leftrightarrow$ definition of PR

$@u \cdot (g(u) \Rightarrow \mathsf{skip}) \sqsubseteq @v \cdot (g(v) \Rightarrow u := v)$

$\Leftrightarrow$ definition of @, $\forall q \in s$

$\bigcap_{u \mid u \in E} (g(u) \Rightarrow \mathsf{skip})(q) \subseteq \bigcap_{v \mid v \in E} g(v) \Rightarrow u := v)(q)$

$\Leftrightarrow$ definition of $\Rightarrow$

$\bigcap_{u \mid u \in E \wedge g(u)} q \in \bigcap_{v \mid v \in E \wedge g(v)} (u := v)(q)$

$\Leftrightarrow$ set transformers theory

$\bigcap_{u \mid u \in E \wedge g(u)} q \in \bigcap_{v \mid v \in E \wedge g(v)} q$

$\Leftrightarrow$ set theory

true

$g(u) \Rightarrow \mathsf{skip} \preceq g(v) \Rightarrow u := v$ (b)

$P \sqsubseteq \neg g(u) \Rightarrow u :\in E \sqsubseteq g(u) \Rightarrow \mathsf{skip} \mathrel{\hat{\preceq}} P \sqsubseteq \neg g(v) \Rightarrow v :\in E \sqsubseteq g(v) \Rightarrow u := v \Leftarrow$

monotonicity $(\neg g(u) \Rightarrow u :\in E \preceq \neg g(v) \Rightarrow v :\in E) \wedge (g(u) \Rightarrow \mathsf{skip} \preceq g(v) \Rightarrow u := v) \Leftrightarrow$ (a) and (b)

true

□

Proof of theorem 6.

$P \sqsubseteq u : g(u) \mathrel{\hat{\preceq}} P \sqsubseteq \neg g(v) \Rightarrow v :\in E \sqsubseteq g(v) \Rightarrow u := v$

$\Leftarrow$ lemma 11

$P \sqsubseteq g(u) \Rightarrow u :\in E \mathrel{\hat{\preceq}} P \sqsubseteq g(v) \Rightarrow v :\in E \sqsubseteq g(v) \Rightarrow u := v$

$\Leftarrow$ theorem 1

$P \sqsubseteq \neg g(u) \Rightarrow u :\in E \sqsubseteq \mathsf{skip} \mathrel{\hat{\preceq}} P \sqsubseteq \neg g(v) \Rightarrow v :\in E \sqsubseteq g(v) \Rightarrow u := v$

$\Leftarrow$ lemma 12

$P \sqsubseteq \neg g(u) \Rightarrow u :\in E \sqsubseteq g(u) \Rightarrow \mathsf{skip} \mathrel{\hat{\preceq}} P \sqsubseteq \neg g(v) \Rightarrow v :\in E \sqsubseteq g(v) \Rightarrow u := v$

$\Leftrightarrow$ lemma 13

true

□