Event Based Sequential Program Development: Application to Constructing a Pointer Program

Jean-Raymond Abrial

ETH, Zurich
jabrial@inf.ethz.ch

Abstract. In this article, I present an "event approach" used to formally develop sequential programs. It is based on the formalism of Action Systems [6] (and Guarded Commands[7]), which is encoded within B [2]. This approach has already been used to develop distributed programs [4]. In the first part, I present the basic concepts. In the second part, I develop a complete example, which is a reshaping of the Shorr and Waite marking technique [1]. This algorithm is interesting because it involves a large number of pointer manipulations.

1 Introduction

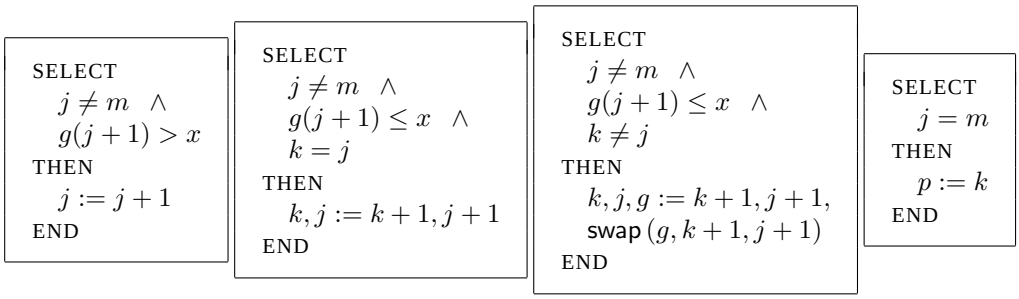
Sequential programs (e.g. loops), when formally constructed, are usually developed gradually by means of a series of progressively more refined "sketches" starting with the formal specification and ending in the final program. Each such sketch is already (although often in a highly non-deterministic form) a monolithic description which resumes the final intended program in terms of a single formula. This is precisely that initial "formula", which is gradually transformed into the final program.

It is argued here that this might *not be the right approach*. After all, in order to prove a large formula, a logician usually breaks it down into various pieces, on which he performs some simple manipulations before putting them again together in a final proof. We would like to experiment with such a paradigm and thus possibly decide whether it is applicable to construct programs as well.

A sequential program is essentially made up of a number of individual assignments that are glued together by means of various constructs: sequential composition (**;**), loop (WHILE) and condition (IF), whose rôle is to explicitly *schedule* these assignments in a proper order so that the execution of the program can achieve its intended goal. Here is an example of a sequential program where the various assignments have been emphasized:

WHILE $j \neq m$ DO
  IF $g(j+1) > x$ THEN
    $\boxed{j := j+1}$
  ELSIF $k = j$ THEN
    $\boxed{k, j := k+1, j+1}$
  ELSE
    $\boxed{k, j, g := k+1, j+1, \mathsf{swap}\,(g, k+1, j+1)}$
  END
END **;**
$\boxed{p := k}$

The idea we want to explore here is to completely separate, during the design, these individual assignments from their scheduling. This approach is thus essentially one by which we favor an initial implicit *distribution of computation* over a centralized explicit one. At a certain stage, the "program" is just made of a number of "naked" guarded commands (which we call here "events"), performing some actions under the control of certain guarding conditions. And at this point the synchronization of these events is not our concern. Thinking operationally, it is done *implicitly* by a hidden scheduler, which *may fire* an event once its guard holds. We can express as follows the various "naked" events corresponding to the previous example (the guard of each event is introduced by the keyword SELECT):



At the beginning of the development process, the event system is made of a single non-guarded event, which represents the specification of our future program. During the development process, other events might be added, which will be further refined together with the initial event. This is done in a certain disciplined manner as shown below. When all the individual pieces are "on the table" (this is the situation shown in the example), and only then, we start to be interested in their *explicit* scheduling. For this, we apply certain systematic rules whose rôle is to gradually *merge the events* and thus organize them into a single entity forming our final program. The application of these rules has the effect of gradually eliminating the various predicates making the guards. At the end of the development, it results in a single guardless final event (as the initial one).

What is interesting about this approach is that it gives us full freedom to refine small pieces of the future program, and also to create new ones, *without being disturbed by others* : the program is developed by means of small *independent* parts that so remain until they are eventually put together systematically at the end of the process.

The paper is organized in two parts. In the first one (section 2), the general framework of event systems is presented. In the second part (section 3) a complete example (inspired by the marking algorithm of Shorr and Waite) is presented.

2 Event System Concepts

2.1 Definition of an Event System: State and Events

An event system is first made of a *state*, which is defined by means of *constants* and *variables*. In practical terms, these constants and variables mainly show simple mathematical objects: sets, binary relations, functions, numbers etc. Moreover, they are constrained by some conditions expressing the *invariant properties* of the system.

Besides its state, an event system contains a number of *events* which show the way it may evolve. Each event is composed of a *guard* and an *action*. The guard is the necessary condition under which

the event may occur. The action, as its name indicates, determines the way in which the state variables are going to evolve when the event occurs.

Once its guard holds, the occurrence of an event may be observed at any time (but may also never be observed). As soon as the guard does not hold however, the event cannot be observed. Events are *atomic* and when the guards of several events hold simultaneously then *at most one of them* may be observed. The choice of the elected event is non-deterministic. Practically speaking, an event, named xxx, is presented in one of the two following simple forms:

$$\boxed{\text{xxx} \;\widehat{=}\; \text{SELECT}\ P(a,b,\ldots)\ \text{THEN}\ S(a,b,\ldots)\ \text{END}} \qquad \boxed{\text{xxx} \;\widehat{=}\; \text{BEGIN}\ S(a,b,\ldots)\ \text{END}}$$

where $P(a,b,\ldots)$ is a predicate denoting the guard, and $S(a,b,\ldots)$ is the action. The list $a,b,\ldots$ denotes some constants and variables of the state. Sometimes, the guard is simply missing and the event may thus take place at any time (this corresponds to the second form shown).

The action presents itself in the form of a simultaneous assignment of certain state variables $a,b,\ldots$ by certain expressions $E,F,\ldots$. Such expressions depend upon the state. It is to be noted that those variables which are not mentioned in the list $a,b,\ldots$ do not change. Such an action can be first written in one of the following two simple equivalent forms:

$$\boxed{a,b,\ldots := E,F,\ldots} \qquad\qquad \boxed{a := E \;\;||\;\; b := F \;\;||\;\; \ldots}$$

There exists however a more general form of action, which is the following:

$$\boxed{\text{ANY}\ x,y,\ldots\ \text{WHERE}\ Q(x,y,\ldots,a,b,\ldots)\ \text{THEN}\ S(x,y,\ldots,a,b,\ldots)\ \text{END}}$$

where the identifiers $x,y,\ldots$ denotes some *constants that are local to the event*. These constants are constrained by the predicate $Q(x,y,\ldots,a,b,\ldots)$. The formula $S(x,y,\ldots,a,b,\ldots)$ denotes a simple deterministic action as above (multiple assignment). Notice that this *non-deterministic* action must be *feasible*. In other words, under the guard of the event, the following must hold:

$$\boxed{\exists\,(x,y,\ldots)\cdot Q(x,y,\ldots,a,b,\ldots)}$$

2.2 Consistency of an Event System

Once a system is built, one must prove that it is *consistent*. This is done by proving that each event of the system preserves the invariant. More precisely, it must be proved that the action associated to each event modifies the state variables in such a way that the corresponding new invariant holds under the hypothesis of the former invariant and of the guard of the event. For a system with state variable v, invariant $I(v)$, and an event of the form indicated on the left, the statement to be proved is the one indicated on the right:

$$\boxed{\text{SELECT}\ P(v)\ \text{THEN}\ v := E(v)\ \text{END}} \qquad \boxed{I(v)\ \wedge\ P(v)\ \ \Rightarrow\ \ I(E(v))}$$

2.3 Refining an Event System

Refining an event system consists of refining its state and its events. A concrete system (with regards to a more abstract one) has a state that should be related to that of the abstraction through a, so-called, *abstraction relation*, which is expressed in terms of an invariant $J(v, w)$ connecting the abstract state represented by the variables v and the concrete state represented by the variables w.

Each event of the abstract model is refined to one or more corresponding events of the concrete one. Informally speaking, a concrete event is said to refine its abstraction (1) when the guard of the former is stronger than that of the latter (guard strengthening), (2) and when the connecting invariant is preserved by the conjoined action of both events. In the case of an abstract event (left) and a corresponding concrete event (right) having the forms

$$\boxed{\text{SELECT } P(v) \text{ THEN } v := E(v) \text{ END}} \qquad \boxed{\text{SELECT } Q(w) \text{ THEN } w := F(w) \text{ END}}$$

then the statement to prove is the following (where $I(v)$ is the abstract invariant and $J(v, w)$ is the connecting invariant):

$$\boxed{I(v) \ \wedge \ J(v,w) \ \wedge \ Q(w) \ \ \Rightarrow \ \ P(v) \ \wedge \ J(E(v), F(w))}$$

Moreover, the concrete system must *not deadlock more often* than the abstract one. This is proved by stating that the disjunction of the abstract guards implies that of the concrete ones, formally:

$$\boxed{I(v) \ \wedge \ J(v,w) \ \wedge \ (P_1(v) \ \vee \ \ldots \ \vee \ P_n(v)) \ \ \Rightarrow \ \ Q_1(w) \ \vee \ \ldots \ \vee \ Q_n(w)}$$

where the P_i and Q_i denote the abstract and concrete guards respectively. Note that this statement could be split into n distinct statements.

2.4 Adding New Events in a Refinement

When refining an event system by another one, it is possible to *add new events*. Such events must be proved to refine the dummy event which does nothing (skip) in the abstraction. Moreover, a special proof must be performed, ensuring that the new events cannot collectively take control for ever. For this, a unique *variant expression* must be "decreased" by each new event. In the case of a new event of the following form:

$$\boxed{\text{SELECT } R(w) \text{ THEN } w := G(w) \text{ END}}$$

the following statement has thus to be proved:

$$\boxed{I(v) \ \wedge \ J(v,w) \ \ \Rightarrow \ \ V(w) \in \mathbb{N}}$$

$$\boxed{I(v) \ \wedge \ P(v) \ \wedge \ J(v,w) \ \ \Rightarrow \ \ J(v, G(w)) \ \wedge \ V(G(w)) < V(w)}$$

where $V(w)$ denotes the variant expression considered (here it is a natural number expression, but it can be more elaborate).

2.5 Special Properties of the Event System Used to Develop Sequential Programs

In this section, we shall express the specific properties that an event system used for sequential program development should satisfy. We shall also fix the *style* we shall adopt in our future program development.

(1) *At the beginning of a development*, our event system is first characterized by some $parameters$, say p, which denote some constant "input" of the future program. In other words they will not evolve when the future program is "run". The constant p are declared as follows:

$$p \in S_p \quad \wedge \quad Pre\_condition(p)$$

where S_p denotes the type of the parameters and $Pre\_condition(p)$ denotes a predicate defining a certain condition, which the $parameters$ should satisfy (besides typing, of course). The initial system also has some variables, called here $results$. These variables are typed with S_r as follows:

$$results \in S_r$$

The initial event system contains only one event that can be fired any time: its guard is simply missing (hence it always holds). It involves the $results$ and describes the characteristic properties of the outcome of the future program. Here is the most general form of this event:

aprog $\widehat{=}$
BEGIN
ANY r WHERE $r \in S_r \ \wedge \ Post\_condition(p, r)$ THEN $result := r$ END
END

where S_r denotes the type of the $results$ and $Post\_condition(p, r)$ denotes the final condition, which the program should satisfy. This condition involves the parameters p as well as the $results$ r. The pre- and post- conditions together represent the *specification* of our program.

Notice that the initial system must contain another special event called init, which allows the initial value of $results$ to freely "float" within its type as follows[1];

init $\widehat{=}$ BEGIN $results :\in S_r$ END

(2) *During the development*, we perform various refinements of the initial event system. As a consequence, at each stage of the development, the current event system may contain more variables and more events.
(3) *At the end of the development*, and after applying some merging rules defined in the next section, one should obtain again a single event of the following form:

cprog $\widehat{=}$
BEGIN
$Initialisation$;
$Program$
END

[1] The construct $x :\in s$ is a shorthand for ANY y WHERE $y \in s$ THEN $x := y$ END.

where $Initialisation$ corresponds to the last version of init and $Program$ is the last version of aprog.

2.6 Merging Rules

We essentially have two merging rules, one for defining a conditional statement and the other one for defining a loop statement. Here are these rules:

```
SELECT P ∧ Q THEN S END
SELECT P ∧ ¬Q THEN T END
⇝
SELECT P THEN
  IF Q THEN S ELSE T END
END
```

```
SELECT P ∧ Q THEN S END
SELECT P ∧ ¬Q THEN T END
⇝
SELECT P THEN
  WHILE Q DO S END ; T
END
```

These rules can be read as follows: if we have an event system where two events have forms corresponding to the ones shown in the antecedent of the rule, they can be merged into a single event corresponding to the consequent of the rule. Notice that both rules have the same "antecedent-events", so that the application of one or the other might be problematic. There is no confusion however as the rules have some *incompatible side conditions*:

– The second rule (that introducing WHILE) requires that the first antecedent event (that giving rise to the "body" S of the loop) appears at *one refinement level below that of the second one*. In this way, we are certain that there exists a variant ensuring that the loop terminates (see section 2.4). Moreover, *the first event must keep the common condition P invariant*. The merging event is considered to "appear" at the same level as the second antecedent event.

– The first rule (that introducing IF) is applicable when the second one is not. The merging event is considered to bear the same "level" as the smallest one. When the two merged events are not at the same level, the "merged variant" becomes the pair of both variants, which thus *decreases lexicographically*.

Note that in both rules, the common guard P is optional. If missing, the application of the rule results in a non-guarded event. The first rule may take a special form when one of the antecedent events has an IF form. It goes as follows:

```
SELECT P ∧ Q THEN S END
SELECT P ∧ ¬Q THEN
  IF R THEN T ELSE U END
END
⇝
SELECT P THEN
  IF Q THEN S ELSIF R THEN T ELSE U END
END
```

3 Example

The example we present in this section is inspired by the marking algorithm of Shorr and Waite. This algorithm has received a considerable attention in the literature, so that it is impossible to cite

all references on the subject (a recent and interesting one is that of R. Bornat [5]). Given a graph and a certain point in it (called the "top"), the marking algorithm computes the image of this point under the transitive closure of the relation defining the graph. Informally, this algorithm is characterized by three properties:

1. It is is a graph traversal algorithm from the top.
2. The traversal is "depth-first".
3. The backtracking structure is stored within the graph itself.

3.1 A One Shot Specification

Let the graph be defined by a constant binary relation g built on a set N of nodes. Let c be the transitive closure of g (the required properties of c will appear in the next section). Let t be any node. The result r is a subset of N. The event mark computes in one shot the image of $\{t\}$ under c. **Fig.1** shows this marking performed in one shot[2].

Constants
$g \in N \leftrightarrow N$
$c \in N \leftrightarrow N$
$t \in N$

Invariant
$r \subseteq N$

Events
init $\widehat{=}$ BEGIN $r :\in \mathbb{P}(N)$ END
mark $\widehat{=}$ BEGIN $r := c[\{t\}]$ END

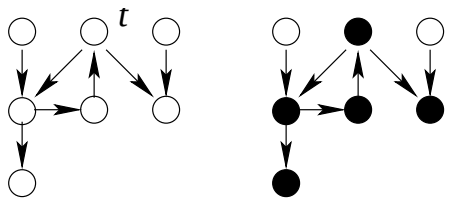


Fig.1. Marking in one shot

3.2 Refinement 1: A Non-deterministic Loop

In this refinement, we introduce a new variables b (for "black"), which is a set of nodes, and a new event called prg1 (for "progress1"). The image of the set $\{t\}$ under the transitive closure c is now computed gradually. The node t is supposed to be in b, and the set b is supposed to be included in $c[\{t\}]$. It is set to the singleton $\{t\}$ in the init event[3]. The guard of event prg1 states that $g[b] - b$ is not empty. An element y of this set is thus chosen arbitrarily and put into the set b. The event mark is now guarded by the condition $g[b] \subseteq b$: in this case the closure $c[\{t\}]$ is exactly equal to the set b. **Fig.2** shows an animation of this non-deterministic algorithm.

[2] The main set-theoretic notations are summarized in the Appendix.
[3] In the event init, we have removed the initialisation of r in order to ease the reading.

Invariant
$b \subseteq N$
$t \in b$
$b \subseteq c\,[\{t\}]$

Events
init $\;\widehat{=}\;$ BEGIN $b := \{t\}$ END
prg1 $\;\widehat{=}\;$ SELECT $g[b] \not\subseteq b$ THEN ANY y WHERE $y \in g[b] - b$ THEN $b := b \cup \{y\}$ END END
mark $\;\widehat{=}\;$ SELECT $g[b] \subseteq b$ THEN $r := b$ END

In order to validate this refinement, we mainly need to prove that (1) the initialisation establishes the invariant $b \subseteq c[\{t\}]$, (2) the event prg1 maintains it, and (3) the concrete version of event mark refines its abstraction. After some elementary transformations, these amount to proving respectively:

To be proved
$\{t\} \subseteq c[\{t\}]$
$x, y \in g \;\wedge\; x \in c[\{t\}] \;\Rightarrow\; y \in c[\{t\}]$
$t \in b \;\wedge\; b \subseteq c[\{t\}] \;\wedge\; g[b] \subseteq b \;\Rightarrow\; c[\{t\}] \subseteq b$

For proving this, we only need the following well known properties of the closure c of g:

Properties of closure c
$\forall s \cdot (\, s \subseteq N \;\Rightarrow\; s \subseteq c[s]\,)$
$\forall\,(s, x, y) \cdot (\, s \subseteq N \;\wedge\; x, y \in g \;\wedge\; x \in c[s] \;\Rightarrow\; y \in c[s]\,)$
$\forall s \cdot (\, s \subseteq N \;\wedge\; g[s] \subseteq s \;\Rightarrow\; c[s] \subseteq s\,)$

We also need to prove that (1) event prg1 refines skip (obvious since it only involves the new variable b), (2) the event system does not deadlock as the abstraction does not (obvious since the disjunction of the guards of prg1 and mark is clearly true), and (3) that the event prg1 decreases some natural number quantity (take the cardinality of the set $N - b$).

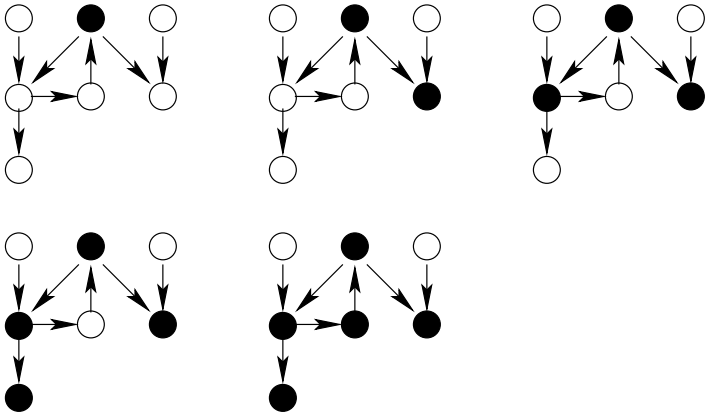


Fig.2. A non-deterministic marking algorithm

3.3 Refinement 2: Making the Loop More Deterministic (Depth-first Marking)

This refinement contains the *first key decision* of the development. The idea is to constrain the previous non-deterministic algorithm to always move more deeply in the graph until one encounters either a terminal node or some previously encountered nodes. At this point, the algorithm backtracks to the previously visited node, and from there continues to explore the graph if possible, and so on. Note that there remains some non-determinacy in this algorithm as the choice of the branch to follow in a node is arbitrary. We introduce three variables in this refinement:

- First a, so-called, *current pointer* p. It always corresponds to a black node, from which we move to a deeper node in the graph. Initially p is set to the "top" node t.
- The second variable is a, so-called, *backtracking structure* f. It allows one to make p revisit the previous node when it cannot pursue further its depth-first graph traversal. The backtracking structure f has some interesting properties as shown in **Fig.3**: it is an injective function, it is made of black nodes only, its domain extended with $\{t\}$ is exactly its range extended with $\{p\}$, it has no cycle, and, when reversed, it is included in the graph g. Moreover, if we consider the image under g of the black nodes that are not in the backtracking structure, this image is made of black nodes only.
- The third variable is a *boolean* n which is used to detect the end of the loop. When equal to OK then $g[b]$ is included in b.

Fig.3. The backtracking Structure

Invariant
$p \in b$
$f \in (b \cup \{p\}) - \{t\} \rightarrowtail\!\!\!\!\!\!\!\!\rightarrow\; (b \cup \{t\}) - \{p\}$

$\mathsf{dom}(f) \cup \{t\} = \mathsf{ran}(f) \cup \{p\}$
$\forall s \cdot (s \subseteq \mathsf{dom}(f) \cup \{t\} \ \wedge \ t \in s \ \wedge \ f^{-1}[s] \subseteq s \ \Rightarrow \ \mathsf{dom}(f) \cup \{t\} \subseteq s)$
$f^{-1} \subseteq g$
$g[b - (\mathsf{dom}(f) \cup \{t\})] \subseteq b$
$n \in \{OK, KO\}$
$n = OK \ \Rightarrow \ g[b] \subseteq b$

Two new events are introduced: prg2 and prg3. Event prg2 is doing the backtracking and event prg3 is detecting the end of the loop (when p is equal to t and when there are no further nodes to explore from t). These events must decrease some natural number quantity (take the cardinality of f augmented with the encoding of n: 1 when KO, 0 when OK). Here are the events of this refinement:

Events
init $\widehat{=}$ BEGIN $b, p, f, n := \{t\}, t, \emptyset, KO$ END
prg1 $\widehat{=}$ SELECT $n = KO \ \wedge \ g[\{p\}] \not\subseteq b$ THEN ANY y WHERE $y \in g[\{p\}] - b$ THEN $b, p, f := b \cup \{y\}, y, f \cup \{y \mapsto p\}$ END END
prg2 $\widehat{=}$ SELECT $n = KO \ \wedge \ g[\{p\}] \subseteq b \ \wedge \ p \neq t$ THEN $p, f := f(p), \{p\} \lhd f$ END
prg3 $\widehat{=}$ SELECT $n = KO \ \wedge \ g[\{p\}] \subseteq b \ \wedge \ p = t$ THEN $n := OK$ END
mark $\widehat{=}$ SELECT $n = OK$ THEN $r := b$ END

At this point, it might be interesting to apply the merging rules, just to have an idea of the abstract program we could obtain. Merging events prg2 and prg3 leads to the following:

prg2_3 $\widehat{=}$
SELECT $n = KO \ \wedge \ g[\{p\}] \subseteq b$ THEN
IF $p \neq t$ THEN
$\boxed{p, f := f(p), \{p\} \lhd f}$
ELSE
$\boxed{n := OK}$
END
END

Now merging events prg1 and prg2_3 leads to the following (note that the WHILE merging rule is not applicable here since event prg2_3, which could be a potential candidate for the loop body since it appears one level below that of prg1, prg2_3 does not keep invariant the common guard $n = KO$):

```
prg1_2_3 ≙
  SELECT n = KO THEN
    IF g[{p}] ⊈ b THEN
      ANY y WHERE y ∈ g[{p}] − b THEN b, p, f := b ∪ {y}, y, f ∪ {y ↦ p} END
    ELSIF q ≠ t THEN
      p, f := f(p), {p} ⩤ f
    ELSE
      n := OK
    END
  END
```

Merging finally events mark and prg1_2_3 leads to the following (we have no problem here applying the WHILE loop since there is no common remaining guard):

```
mark_prg1_2_3 ≙
  WHILE n = KO DO
    IF g[{p}] ⊈ b THEN
      ANY y WHERE y ∈ g[{p}] − b THEN b, p, f := b ∪ {y}, y, f ∪ {y ↦ p} END
    ELSIF q ≠ t THEN
      p, f := f(p), {p} ⩤ f
    ELSE
      n := OK
    END
  END ;
  r := b
```

By adding the init event, we obtain the following abstract program:

```
b, p, f, n := {t}, t, ∅, KO ;
WHILE n = KO DO
  IF g[{p}] ⊈ b THEN
    ANY y WHERE y ∈ g[{p}] − b THEN b, p, f := b ∪ {y}, y, f ∪ {y ↦ p} END
  ELSIF q ≠ t THEN
    p, f := f(p), {p} ⩤ f
  ELSE
    n := OK
  END
END ;
r := b
```

In **Fig.4**, you can see an animation of this abstract algorithm. Notice the current pointer (emphasized in grey, but it is also black!) and the backtracking structure situated next to the pointers forming the graph.

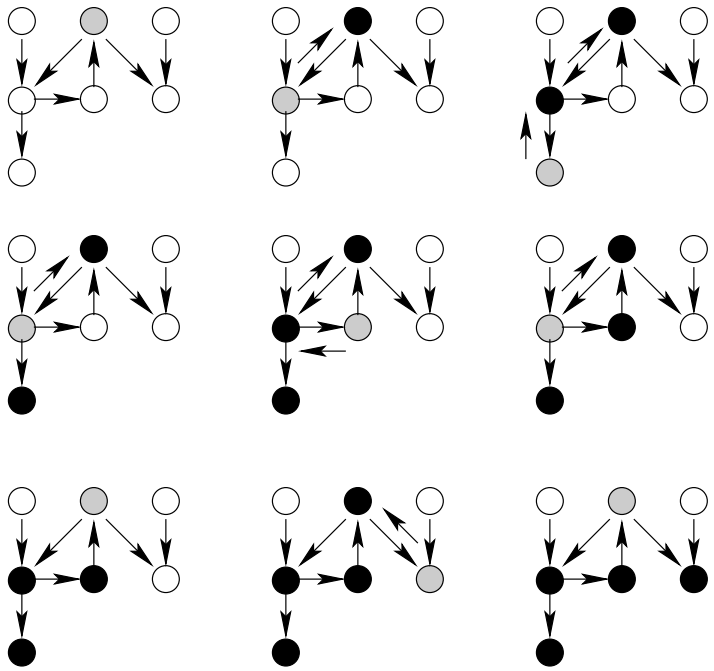


Fig.4. A depth-first marking with backtracking structure and current point

3.4 Refinement 3: Specializing the Graph by Means of Two Partial Functions

In this refinement, we refine the constant relation g by two constant partial functions called lt (for 'left") and rt (for "right"). There are no changes in the variables. The event prg1 is refined by two events prg11 and prg12. They correspond to a left or right descent respectively. The algorithm is now completely deterministic: one first tries to move down along the left path then along the right one. **Fig.5.** shows an animation of this algorithm (we have represented the left arrows in white and the right ones in black).

Events
init $\widehat{=}$ BEGIN $b, p, f, n := \{t\}, t, \emptyset, KO$ END
prg11 $\widehat{=}$ SELECT $n = KO \ \wedge \ lt[\{p\}] \not\subseteq b$ THEN $b, p, f := b \cup \{lt(p)\}, lt(p), f \cup \{lt(p) \mapsto p\}$ END
prg12 $\widehat{=}$ SELECT $n = KO \ \wedge \ lt[\{p\}] \subseteq b \ \wedge \ rt[\{p\}] \not\subseteq b$ THEN ... END
prg2 $\widehat{=}$ SELECT $n = KO \ \wedge \ g[\{p\}] \subseteq b \ \wedge \ p \neq t$ THEN $p, f := f(p), \{p\} \lhd f$ END
prg3 $\widehat{=}$ SELECT $n = KO \ \wedge \ g[\{p\}] \subseteq b \ \wedge \ p = t$ THEN $n := OK$ END
mark $\widehat{=}$ SELECT $n = OK$ THEN $r := b$ END

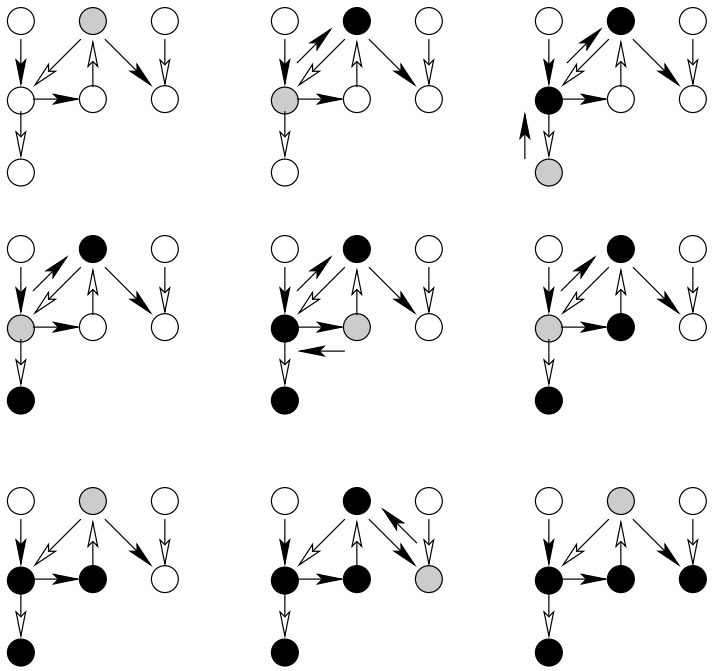


Fig.5. Adding left and right pointers

3.5 Refinement 4: Decorating the Nodes of the Backtracking Structure

In this refinement we decorate each node of the range of the backtracking structure (thus p is not decorated). The intent of decorating a node is to record the fact that the path followed by the algorithm from that node was the left or the right one. On **Fig.6** you can see that some nodes of the range of f are decorated with "l" (for left) and another one with "r" (for right). When the decoration is "l", it means that the algorithm has chosen the left pointer in doing further visits from this node, and similarly for the other direction. We have painted in white or black the pointer of the backtracking structure whose initial node is decorated "l" or "r" respectively.

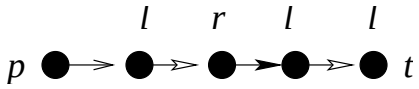


Fig.6. Decorating the nodes (except first) and pointers (except first) of the backtracking structure f

We introduce two variables, called lft and rht, corresponding to the sets of decorated nodes. These sets form a partition of the range of f. Moreover by reversing a pointer of f which is ending in a node of lft we obtain a pointer of lt and similarly for the other direction. These properties are formalized in the following table:

Invariant
$lft \subseteq N$
$rht \subseteq N$
$lft \cup rht = \mathsf{ran}\,(f)$

Invariant
$lft \cap rht = \emptyset$
$lft \lhd f^{-1} \subseteq lt$
$rht \lhd f^{-1} \subseteq rt$

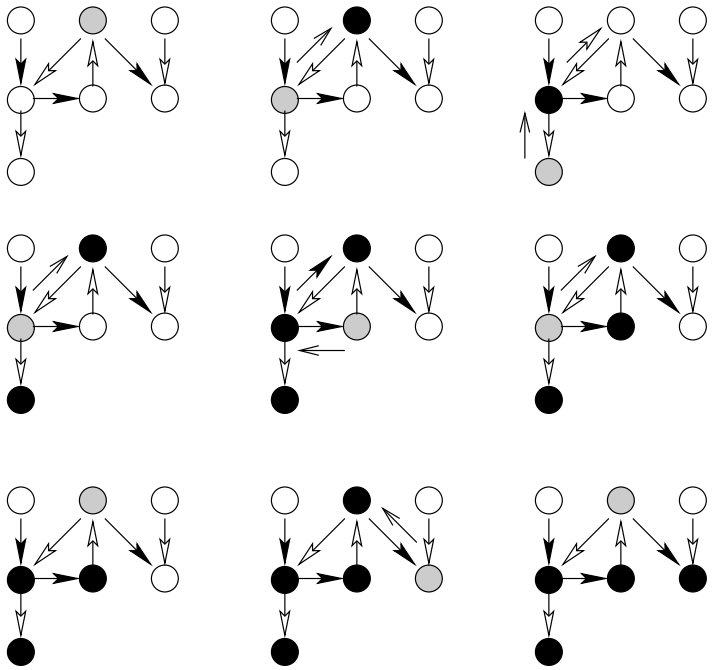


Fig.7. Decorating the backtracking structure

The event prg2 is refined by two events prg21 and prg22. They correspond to a left or right backtracking respectively. Here are the events of this refinement:

Events
init $\widehat{=}$ BEGIN $b, p, f, n, lft, rht := \{t\}, t, \emptyset, KO, \emptyset, \emptyset$ END
prg11 $\widehat{=}$ SELECT $n = KO \ \wedge \ lt[\{p\}] \not\subseteq b$ THEN $b, p, f, lft := b \cup \{lt(p)\}, lt(p), f \cup \{lt(p) \mapsto p\}, lft \cup \{p\}$ END
prg12 $\widehat{=}$ ...

prg21 $\widehat{=}$ SELECT $n = KO \ \wedge \ g[\{p\}] \subseteq b \ \wedge \ p \neq t \ \wedge \ f(p) \in lft$ THEN $p, f, lft := f(p), \{p\} \lhd f, lft - \{f(p)\}$ END
prg22 $\widehat{=}$ ...
prg3 $\widehat{=}$ SELECT $n = KO \ \wedge \ g[\{p\}] \subseteq b \ \wedge \ p = t$ THEN $n := OK$ END
mark $\widehat{=}$ SELECT $n = OK$ THEN $r := b$ END

3.6 Refinement 5: Storing Part of Backtracking Structure Within the Graph

This refinement contains the *second key decision* of the development. This is the main idea of the Schorr and Waite paper: it consists in storing the backtracking structure in the graph itself, which is thus modified during the execution but recovers its initial setting at the end of it.

For this, we introduce two new variables called ult and urt. They represent the "dynamic" left and right pointers of the graph. The backtracking structure is now *almost* stored in ult and urt, "almost" because there remains the pair $p \mapsto f(p)$, which, when it exists, cannot be stored in the graph. For this, we define a new variable called h: it is a mini-function which is either empty or contains a single pair. Here are the definitions and properties of these variables:

Invariant
$ult \in N \nrightarrow N$
$urt \in N \nrightarrow N$
$h \in N \nrightarrow N$

Invariant
$ult = lft \lhd lt \ \cup \ lft \lhd f$
$urt = rht \lhd rt \ \cup \ rht \lhd f$
$h = \{p\} \lhd f$

Events
init $\widehat{=}$ BEGIN $b, p, f, n, lft, rht, ult, urt := \{t\}, t, \emptyset, KO, \emptyset, \emptyset, lt, rt$ END
prg11 $\widehat{=}$ SELECT $n = KO \ \wedge \ ult[\{p\}] \not\subseteq b$ THEN $b, p, h, lft, ult := b \cup \{ult(p)\}, ult(p), (\{p\} \lhd ult)^{-1}, lft \cup \{p\}, \{p\} \lhd ult \cup h$ END
prg12 $\widehat{=}$ ...

prg21 $\widehat{=}$ SELECT $n = KO \ \wedge \ (ult \cup urt)[\{p\}] \subseteq b \ \wedge \ p \neq t \ \wedge \ h(p) \in lft$ THEN $p, h, lft := h(p), \{h(p)\} \lhd ult, lft - \{h(p)\} \parallel$ $ult(h(p)) := p$ END
prg22 $\widehat{=}$ ...
prg3 $\widehat{=}$ SELECT $n = KO \ \wedge \ (ult \cup urt)[\{p\}] \subseteq b \ \wedge \ p = t$ THEN $n := OK$ END
mark $\widehat{=}$ SELECT $n = OK$ THEN $r := b$ END

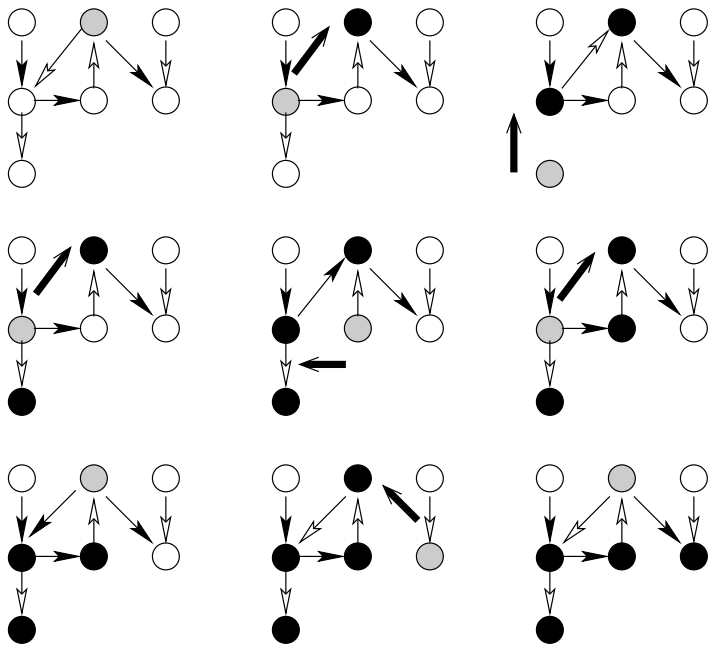


Fig.8. Storing part of the backtracking structure within the graph (the remaining part is emphasized)

3.7 Refinement 6: Introducing the Dummy Node "nil" and a Second "Current" Pointer

In this refinement, we implement the mini-function h which, when non-empty, is made of a unique pair starting in p. The implementation is done by means of a second "sub-current" pointer q. In fact, when it is empty, then p is situated at the top, t. In order to implement the pair $p \mapsto h(p)$ even at the point t, we have no choice but introducing a "dummy" node, called nil. This is the purpose of this refinement.

This leads to a simple modification of the dynamic pointers ult and urt, which becomes vlt and vrt. The new pointer is called q. Here are the formal definitions and properties of these new variables:

Constants
$nil \in N$
$nil \notin \mathsf{dom}(lt) \cup \mathsf{dom}(rt)$
$nil \notin \mathsf{ran}(lt) \cup \mathsf{ran}(rt)$
$t \neq nil$

Invariant
$vlt \in N \nrightarrow N$
$vrt \in N \nrightarrow N$
$q \in N$

Invariant
$vlt = ult \lhd\!\!+ (lft \lhd \{t \mapsto nil\})$
$vrt = urt \lhd\!\!+ (rht \lhd \{t \mapsto nil\})$
$p = t \;\Rightarrow\; q = nil$
$p \neq t \;\Rightarrow\; q = h(t)$

Events
init $\;\widehat{=}\;$ BEGIN $b, p, q, n, lft, rht, vlt, vrt := \{t\}, t, nil, KO, \emptyset, \emptyset, lt, rt$ END
prg11 $\;\widehat{=}\;$ SELECT $n = KO \;\wedge\; vlt[\{p\}] \not\subseteq b$ THEN $b, p, q, lft := b \cup \{vlt(p)\}, vlt(p), p, lft \cup \{p\} \parallel vlt(p) := q$ END
prg12 $\;\widehat{=}\;$ ...
prg21 $\;\widehat{=}\;$ SELECT $n = KO \;\wedge\; (vlt \cup vrt)[\{p\}] \subseteq b \;\wedge\; p \neq t \;\wedge\; q \in lft$ THEN $p, q, lft := q, vlt(q), lft - \{q\} \parallel vlt(q) := p$ END
prg22 $\;\widehat{=}\;$ ...
prg3 $\;\widehat{=}\;$ SELECT $n = KO \;\wedge\; (vlt \cup vrt)[\{p\}] \subseteq b \;\wedge\; p = t$ THEN $n := OK$ END
mark $\;\widehat{=}\;$ SELECT $n = OK$ THEN $r := b$ END

3.8 Refinement 7: Making the Graph Functions Total

In this refinement we are making the two functions vlt and vrt total by introducing nil as a dummy value when they are not defined. For this we define two new constants ltp and rtp and two new variables wlt and wrt. They are as follows:

Constants
$ltp \in N \rightarrow N$
$rtp \in N \rightarrow N$
$ltp = (N \times \{nil\}) \lhd lt$
$rtp = (N \times \{nil\} \lhd rt$

Invariant
$wlt \in N \rightarrow N$
$wrt \in N \rightarrow N$
$wlt = (N \times \{nil\}) \lhd vlt$
$wrt = (N \times \{nil\} \lhd vrt$

init $\widehat{=}$ BEGIN $b, p, q, n, lft, rht, wlt, wrt := \{t\}, t, nil, KO, \emptyset, \emptyset, ltp, rtp$ END
prg11 $\widehat{=}$ SELECT $n = KO \ \wedge \ wlt(p) \neq nil \ \wedge \ wlt(p) \notin b$ THEN $b, p, q, lft := b \cup \{wlt(p)\}, wlt(p), p, lft \cup \{p\} \parallel wlt(p) := q$ END
prg12 $\widehat{=}$ ...
prg21 $\widehat{=}$ SELECT $n = KO \ \wedge \ \neg (wlt(p) \neq nil \ \wedge \ wlt(p) \notin b) \ \wedge$ $\neg (wrt(p) \neq nil \ \wedge \ wrt(p) \notin b) \ \wedge \ p \neq t \ \wedge \ q \in lft$ THEN $p, q, lft := q, wlt(q), lft - \{q\} \parallel wlt(q) := p$ END
prg22 $\widehat{=}$ ...
prg3 $\widehat{=}$ SELECT $n = KO \ \wedge \ \neg (wlt(p) \neq nil \ \wedge \ wlt(p) \notin b) \ \wedge$ $\neg (wrt(p) \neq nil \ \wedge \ wrt(p) \notin b) \ \wedge \ p = t$ THEN $n := OK$ END
mark $\widehat{=}$ SELECT $n = OK$ THEN $r := b$ END

3.9 Refinement 8: Introducing Color and Direction

In this last refinement we encode the sets b, lft and rht. The first one, b, is encoded by means of a total function clr capable of taking two values, namely $BLACK$ and $WHITE$. The second and third sets are together encoded by means of a total function dir capable of taking two values, namely $LEFT$ and $RIGHT$. The verification of this refinement requires to prove that the pointer q is either in the set lft or in the set rht. Here are the definitions of these new variables:

Invariant
$clr \in NODE \to \{BLACK, WHITE\}$
$dir \in NODE \to \{LEFT, RIGHT\}$

Invariant
$b = clr^{-1}[\{BLACK\}]$
$lft \subseteq dir^{-1}[\{LEFT\}]$
$rht \subseteq dir^{-1}[\{RIGHT\}]$

init $\widehat{=}$ BEGIN $p, q, n := t, nil, KO \parallel clr(t) := BLACK$ END
prg11 $\widehat{=}$ SELECT $n = KO \ \wedge \ wlt(p) \neq nil \ \wedge \ clr(wlt(p)) = WHITE$ THEN $clr(wlt(p)) := BLACK \parallel p, q := wlt(p), p \parallel$ $dir(p) := LEFT \parallel wlt(p) := q$ END
prg12 $\widehat{=}$ SELECT $n = KO \ \wedge \ \neg(wlt(p) \neq nil \ \wedge \ clr(wlt(p)) = WHITE) \ \wedge$ $wrt(p) \neq nil \ \wedge \ clr(wrt(p)) = WHITE$ THEN $clr(wrt(p)) := BLACK \parallel p, q := wrt(p), p \parallel$ $dir(p) := RIGHT \parallel wrt(p) := q$ END
prg21 $\widehat{=}$ SELECT $n = KO \ \wedge \ \neg(wlt(p) \neq nil \ \wedge \ clr(wlt(p)) = WHITE) \ \wedge$ $\neg(wrt(p) \neq nil \ \wedge \ clr(wrt(p)) = WHITE) \ \wedge \ p \neq t \ \wedge \ q \in lft$ THEN $p, q := q, wlt(q) \parallel wlt(q) := p$ END
prg22 $\widehat{=}$ SELECT $n = KO \ \wedge \ \neg(wlt(p) \neq nil \ \wedge \ clr(wlt(p)) = WHITE) \ \wedge$ $\neg(wrt(p) \neq nil \ \wedge \ clr(wrt(p)) = WHITE) \ \wedge \ p \neq t \ \wedge \ q \notin lft$ THEN $p, q := q, wrt(q) \parallel wrt(q) := p$ END
prg3 $\widehat{=}$ SELECT $n = KO \ \wedge \ \neg(wlt(p) \neq nil \ \wedge \ clr(wlt(p)) = WHITE) \ \wedge$ $\neg(wrt(p) \neq nil \ \wedge \ clr(wrt(p)) = WHITE) \ \wedge \ p = t$ THEN $n := OK$ END
mark $\widehat{=}$ SELECT $n = OK$ THEN $r := clr^{-1}[\{BLACK\}]$ END

3.10 Obtaining the Final Program

Applying the merging rules in very much the same way as we have done it in section 3.3, leads to the following program. Note that we have not mentioned the mark event at the end of the program since our final purpose is just to change the colors of the relevant nodes. Also note that this program needs a few more obvious transformations in order to remove the various occurrences of the "$\|$" operators and replace them by "**;**". This would require the introduction of some local variables, and can be done on a purely syntactic basis.

```
p, q, n := t, nil, KO ||
clr(t) := BLACK
;
WHILE n = KO DO
  IF wlt(p) ≠ nil ∧ clr(wlt(p)) = WHITE THEN
    clr(wlt(p)) := BLACK ||
    p, q := wlt(p), p ||
    dir(p) := LEFT ||
    wlt(p) := q
  ELSIF wrt(p) ≠ nil ∧ clr(wrt(p)) = WHITE THEN
    clr(wrt(p)) := BLACK ||
    p, q := wrt(p), p ||
    dir(p) := RIGHT ||
    wrt(p) := q
  ELSIF p ≠ t THEN
    IF dir(q) = LEFT THEN
      p, q := q, wlt(q) || wlt(q) := p
    ELSE
      p, q := q, wrt(q) || wrt(q) := p
    END
  ELSE
    n := OK
  END
END
```

As can be seen, this algorithm is *very symmetric* with respect to the "left" and "right" directions. This was not the case with the original Schorr and Waite algorithm. An interesting outcome of the algorithm presented here is that it seems more efficient than the original one, where each marked node is visited three times. Here each marked node is visited once plus the number of out-going pointers leading to genuine nodes that are not already marked. In the example shown, where we have five marked nodes, the original algorithm takes 15 iterations, whereas the one presented here takes only 9 iterations: two nodes are visited three times and three nodes are visited only once.

3.11 About Proofs

The development has been entirely proven with Atelier B [8], the tool associated with the B Method. In the following table, the statistical details of these proofs can be seen. It shows that approximatively 70% of the proofs are done automatically.

The interactive proofs have all been done with the new interface, "Click'n Prove" [3], which has recently been developed for Atelier B. Most of them are simple. A few of them, however, are

technically (not mathematically) slightly more difficult. You can observe that, by the end of the development a larger proportion of proofs have to be done interactively: this is essentially due to the growing presence of useless hypotheses, which induce some noise and thus make the automatic prover less efficient. The interaction then essentially consists of *reducing the number of relevant hypotheses* and then launching the automatic prover.

Refinement Steps	Automatic	Interactive	Total
1. Non-deterministic Traversal	1	3	4
2. Depth-First Traversal	25	4	29
3. Introducing Left and Right Partial Functions	4	0	4
4. Decorating the Backtracking Structure	13	1	14
5. Storing the Backtracking Structure	24	8	32
6. Introducing “nil” and the Second Pointer	10	11	21
7. Making Left and Right Functions Total	26	13	26
8. Introducing Colors and Directions	12	11	23
TOTAL	108	45	153

4 Conclusion

We have presented an event-based approach to the development of sequential programs, and we have demonstrated it on a well-known non-trivial pointer program: the Schorr-Waite marking algorithm. Contrary to what is done usually, rather than proving it directly, we formally develop this program by means of a number of successive refinement steps. Note that the first two objectives of the Schorr-Waite algorithm (namely graph traversal from “top” and depth-first traversal) are reached after the first and second refinements, whereas the third one (storing the backtracking function in the graph) is obtained after the 5th and 6th refinements.

Acknowledgements

I would like to warmly thank D. Cansell and W. Stoddart for their very careful reading of this text.

References

1. H. Schorr and W.M. Waite. *An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures*. CACM Aug 1967
2. J.R. Abrial. *The B-Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.
3. J.R. Abrial and D. Cansell *Click’n Prove: Interactive Proofs Within Set Theory* TPHOLs 2003.

4. J.R. Abrial, D. Cansell and D. Méry *A Mechanically Proved and Incremental Development of IEEE Tree Identify Protocol* Formal Aspect of Computing 14. 2003
5. R. Bornat. *Proving Pointer Programs in Hoare Logic*. In R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction (MPC 2000)*, LNCS 1837. Springer-Varlag, 2000.
6. R.J.R Back and R. Kurki-Suonio. *Decentralization of Process Nets with Centralized Control* 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. 1983.
7. E.W. Dijkstra *A Discipline of Programming* Prentice-Hall. 1976.
8. Clearsy *ATELIER B. User Manual*. Aix-en-Provence. 2001.

Appendix: Summary of Notations

Notation	Meaning
$\mathbb{P}(s)$	Set of subsets of s
$s \times t$	Cartesian product of s and t
$s \leftrightarrow t$	Set of binary relations from s to t. Same as $\mathbb{P}(s \times t)$
$r[s]$	Image of set s under binary relation r
$s \nrightarrow t$	Set of partial functions from s to t
$s \rightarrowtail t$	Set of partial injections from s to t.
$s \lhd r$	Binary relation r reduced to pairs starting in set s
$s \lhd\!\!\!\!- r$	Binary relation r reduced to pairs not starting in set s
$a \mapsto b$	The pair made of a and b
$f \lhd\!\!+ g$	Relation f overridden by relation g. Same as $(\mathsf{dom}(g) \lhd\!\!\!\!- f) \cup g$
$f(x) := y$	Same as $f := f \lhd\!\!+ \{x \mapsto y\}$