15

The Schorr-Waite-Algorithm

by

Richard Bubel

The Schorr-Waite graph marking algorithm named after its inventors Schorr and Waite [1967] has become an unofficial benchmark for the verification of programs dealing with linked data structures.

It has been originally designed with a LISP garbage collector as application field in mind and thus, its main characteristic is low additional memory consumption. The original design claimed only two markers per data object and, more important, only three auxiliary pointers at all during the algorithm's runtime. It is the latter point, where most other graph marking algorithms lose against Schorr-Waite and need to allocate (often implicitly as part of the method stack) additional memory linear in the number of nodes in the worst case. These resources are used to log the taken path for later backtracking when a circle is detected or a sink reached.

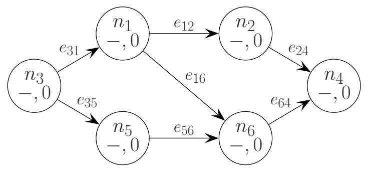
Schorr and Waite's trick is to keep track of the path by reversing traversed edges offset by one and restoring them afterwards in the backtracking phase of the algorithm. A detailed description including the Java implementation to be verified is given in Section 15.1.

Formal treatment of Schorr-Waite is challenging as reachability issues are involved. Transitive closure resp. reachability is beyond pure first-order logic and some extra effort has to be spent to deal with this kind of problems (see [Beckert and Trentelman, 2005] for a detailed discussion). On the other side, the algorithm is small and simple enough to serve as a testbed for different approaches. We introduce a notion of reachability as part of Sect. 15.2 and come back to it for the actual verification, which makes up most of Sect. 15.3.

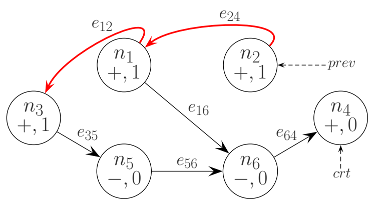
15.1 The Algorithm in Detail

15.1.1 In Theory

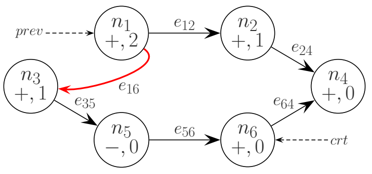
As usual a *directed graph* G is defined as a set of vertices V and edges $E \subseteq V \times V$. The directed edge $s \rightharpoonup t \in E$ connects source node $s \in V$ with target



(a) Initial unmarked Graph



(b) Visiting node n_4 via n_2; edge e_{12} (e_{24}) reverses formerly traversed e_{31} (resp. e_{12})



(c) Visiting node n_6 via n_1; back-tracking restored the formerly modified edges e_{12}, e_{24}

Fig. 15.1. Illustration of a Schorr-Waite run: curved edges have been modified to encode the taken path; pointers *crt*, *prev* refer to the current resp. the previous node

node $t \in V$, but not vice versa. We call node t a direct *successor* of node s (resp. s a direct *predecessor* of t).

For sake of simplicity, we require that each edge e is labelled with a unique natural number $l(e)$ where $l : E \to \mathbb{N}$. The labelling allows us to put an order on all outgoing edges $e_i := s \rightharpoonup_i t_i, i \in \{1, \ldots, n\}$ of a node s, which complies with the natural number ordering $\leq$ of the corresponding labels $l(e_i)$.

When speaking of visiting all children (of a node s) from left-to-right, we mean in fact that all direct successors of s are accessed via its outgoing edges in ascending order of their labels. We refer to the target node of the edge with the i-th smallest label of all outgoing edges of node s as the node's i-th child.

In addition, each node is augmented with a flag `visited` and an integer field `usedEdge`, which is used to store the number of the most recently visited child via this node (or equivalently the corresponding edge label).

In the subsequent four additional pointers are required:

- `current` and `previous`, whose intended purpose is to refer to the currently respective previously visited node and
- the two helpers `next` and `old`.

Given a directed graph G as for example shown in Fig. 15.1 and a designated node s, here: n_3, Schorr-Waite explores G starting at node s applying a left depth-first strategy:

1. Outgoing from the currently visited node `current` the leftmost not yet visited child `next` is selected and the taken edge e redirected to target the node referenced by pointer `previous`. The `usedEdge` field of `current` is used to keep the label $l(e)$ of the reversed edge in order to restore edge e later in the backtracking phase (step 2). Afterwards `previous` is altered to point to our current node, while pointer `current` moves onto node `next`. Finally, the new `current` node becomes marked as visited. Continue with step 1.
2. If all children of the node referred to by `current` have already been visited or it is a childless node and `current` $\neq s$, then a backward step is performed. Therefore the edge via which `current` has been accessed and remembered in the `usedEdge` field of node `previous` during step 1, is restored: this means to redirect it to its original target `current`, but not before rescuing its current target using pointer `old`. Now pointer `current` can be reset to the node referenced by `previous` and—last but not least—`previous` is moved back to node `old`. Continue with step 1.

After all reachable nodes have been visited the algorithm terminates when after a backtracking step the starting node s is reached. At this time the original graph structure has been also restored.

15.1.2 In Practice

The design of our JAVA implementation to be verified is illustrated in Fig. 15.2. The graph nodes are modelled as instances of class `HeapObject`, where each instance contains a `children` array, whose i-th component contains the node's i-th child.

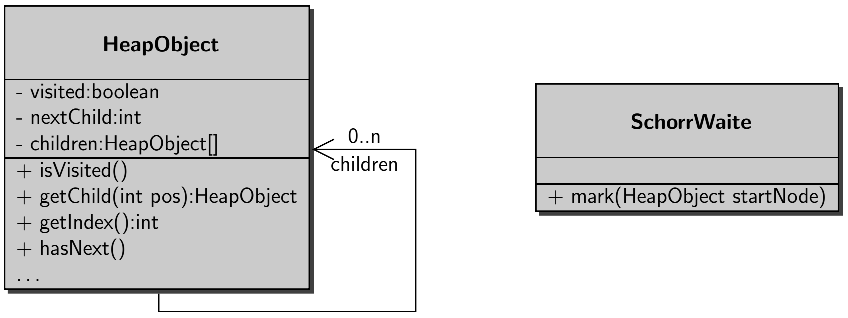


Fig. 15.2. Class diagram showing the involved participants

All `HeapObject` instances provide a rudimentary iterate facility to access their children. Therefore, they implement an integer index field, which contains the array index of the child to be visited next. Method `hasNext` tests if the index field has reached the end of the array and therewith all children of the node have been accessed. The index field is used to realise the `usedEdge` field of the previously given description. In fact, `usedEdge` is equal to the value stored in the index field minus one.

The Java implementation of the algorithm itself is realised as method `mark` of class `Schorr-Waite` shown in Fig. 15.3. Invoking `mark` with a non-null start node handed over as argument starts the graph traversal. The method assumes that before it is invoked, all `HeapObject` instances have no marks set.

```
public void mark(HeapObject startNode) {
  HeapObject current  = start;
  HeapObject previous = null;
  HeapObject next     = null;
  HeapObject old      = null;

  startNode.setMark(true);

  while (current != startNode || startNode.hasNext()) {
    if (current.hasNext()) {
      final int nextChild = current.getIndex();
      next = current.getChild(nextChild);
      if (next != null && !next.isMarked()) {
        // forward scan
        current.setChild(nextChild, previous);
        current.incIndex();
        previous = current;
        current  = next;
        current.setMark(true);
      } else {
        // already visited or no child at this slot
        // proceed to next child
        current.incIndex();
      }
    }  else {
      // backward
      final int ref2restore = previous.getIndex() - 1;
      old = previous.getChild(ref2restore);
      previous.setChild(ref2restore, current);
      current  = previous;
      previous = old;
    }
  } }
```

Fig. 15.3. Core of the Schorr-Waite algorithm

The first lines of method `mark` initialise the required pointers `current`, `previous`, `next` and `old`. The starting node s of the previous section is handed over as the method's argument referred to by parameter `startNode`. It becomes also the first node `current` points to. All other pointers are set to **`null`** at first. Before the while loop is entered, the starting node `startNode` is marked.

After these preparations the graph will be traversed in left depth-first order as long as pointer `current` has not yet returned to the starting node `startNode` or if node `startNode` still has children that need to be checked.

If node `current` has a child left, a forward step is performed (lines 10-25):

line 12 the `nextChild`th component of node `current`'s children array (that is the `current`'s next not already accessed child) is assigned to variable `next`

lines 14–19 these lines are entered in case that the `HeapObject` instance referred to by `next` has not already been visited, which is tested by method `isMarked` in the conditionals guard. First the taken edge, i.e., the `nextChild`th component of `current`'s children array, has to be redirected to the node variable `previous` refers to (line 15). In the succeeding lines, field `nextChild` of node `current` is updated, pointer `previous` is moved toward the node `current` refers to, and finally, node `next` becomes the new `current` node. At the end the new `current` node is marked as visited with help of method `setMark`.

line 23 is only executed in case that node `next` has been already marked in a previous step

Lines 26-32 are executed if node `current` has no remaining children to be visited. In this case a backward step has to be performed:

line 28 the `nextChild-1`th child of node `previous`, which stores the penultimate node is memorised in `old`

line 29 the `nextChild-1`th children array component of node `previous` is restored, i.e., redirected to node `current`

lines 30–31 finally, `current` becomes `previous` and `previous` is set to the node stored in `old`.

15.2 Specifying Schorr-Waite

A correct implementation of Schorr-Waite must guarantee at least these requirements:

First, the algorithm invoked on an arbitrary node of a finite graph must terminate. And second, in its final state it is ensured that

1. When initially invoked on an unmarked graph, a node has been marked if and only if it has been reachable from the starting node.

2. The graph structure has not been modified, i.e., after the algorithm terminates all components of the children arrays contain their original value again.

We concentrate on the specification and verification of the first requirement. The second one is a relative straightforward extension to the first one and can be specified and proven in a similar manner.

15.2.1 Specifying Reachability Properties

Reasoning about linked or recursive data structures requires some notion of reachability of objects. Therefore, we define a reachable predicate that allows to express that an object is reachable from another one via a specified set of fields *Acc*.

In order to express these properties we have to define a variant of non-rigid predicate (function) symbols equipped with a list of locations/accessor expressions that are allowed to be used in order to navigate between objects. We call them *non-rigid symbols with explicit dependencies.*

A New Class of Symbols

A first step in the definition of these symbols, is to define a notion of accessor expressions. You might want to look ahead to Example 15.9 to get an idea what we are aiming at.

Definition 15.1 (Accessor expression). *The set AE of* accessor expressions *is inductively defined as follows:*

1. $(\cdot)_C.\mathtt{a}@(C)$ *is an accessor expression for all attributes* `a` *declared in a class type* C,
2. $f(*, \ldots, *, \overbrace{a}^{i}, *, \ldots, *)$ *is an accessor expression for any arbitrary expression* $a \in AE$*, n-ary function symbol* f*,* $1 \leq i \leq n$ *and the sort of* a *is compatible with the i-th argument sort of* f*.*

Note 15.2. Any accessor expression *acc* contains exactly one placeholder $(\cdot)_C$. We will write *s.acc* when we mean to replace $(\cdot)_C$ by a term s with $sort(s) \leq C$. The suffix $@(C)$ to the attribute `a` only serves to disambiguate attributes with the same name in different classes.

Example 15.3. Let `List` denote a class type declaring an attribute `next` of the same type. Further, let `ASTNode` be another class type declaring an attribute `children` of array type `ASTNode[]`. Then both $(\cdot)_{\mathtt{List}}.\mathtt{next}$ and $(\cdot)_{\mathtt{ASTNode}}.\mathtt{children[*]}$ are accessor expressions. When there is only one obvious choice for the placeholder $(\cdot)_C$ we omit it. We will thus write these two accessor expressions as `next` and `children[*]`.

Definition 15.4 (Syntax and signature: non-rigid symbols with explicit dependencies). *The non-rigid predicate/function symbol* $p[acc\_list;] : T_1 \times \cdots \times T_n$*, resp.,* $f[acc\_list;] : T_1 \times \cdots \times T_n \to T$*, where* acc_list *is a semicolon separated list of accessor expressions are called* non-rigid symbols with explicit dependencies .

Terms and formulas are defined as before(⇒ Sect. 3.2) with the only difference that the corresponding sets of function and predicate symbols may contain these newly introduced symbols.

Definition 15.5 (Semantics). *Let* $\mathcal{K} = (\mathcal{M}, \mathcal{S}, \rho)$ *denote a* JAVA CARD *DL Kripke structure. Then for any two states* S_1 *and* $S_2 \in \mathcal{S}$*, the predicate* $p[acc\_list;](t_1, \ldots, t_n)$ *evaluates to the same truth value, if* S_1 *and* S_2 *coincide on the interpretation of all accessor expressions, i.e., for any* $acc \in acc\_list$ *with placeholder* $(\cdot)_C$*, and for all* $u, v \in Term^C_\Sigma$ *with* $val_{S_1}(u) = val_{S_2}(v)$ *the following holds:* $val_{S_1}(u.acc) = val_{S_2}(v.acc)$*. In case of nested accessor expressions, both states have to coincide on the constituents and in case of an array expression on the* `length` *attribute as well. Analogous for function symbols.*

The Reachable Predicate

The reachable predicate is a representative of a non-rigid predicate with explicit dependencies. Its syntax is defined as follows:

Definition 15.6 (Reachable predicate, syntax). *The ternary non-rigid predicate* $\mathtt{reach}[acc\_list;](T, T, \mathtt{int})$ *is called* reachable predicate*, where* acc_list *is a semicolon separated list of accessor expressions, whose usage is allowed in navigation expressions.*

Example 15.7. Let o, u and s, t be program variables of type `List` resp. `ASTNode` and n be an arbitrary integer constant, then $\mathtt{reach}[next;](\mathtt{o}, \mathtt{o}, 0)$, $\mathtt{reach}[next;](\mathtt{o}, \mathtt{u}, \mathtt{n})$ and $\mathtt{reach}[children[*];](\mathtt{s}, \mathtt{t}, \mathtt{n})$ are syntactical correct JAVA CARD DL formulas.

The semantics of the reachable predicate needs to be defined. The definition has to adhere to the constraint given in Def. 15.5.

Definition 15.8 (Reachable predicate, semantics). *The reachable predicate* $\mathtt{reach}[acc\_list;](\mathtt{o}, \mathtt{u}, \mathtt{n})$ *is valid in a state* s *iff.* $s \models \exists p_1, \ldots, p_m; (o.a_1. \ldots .a_n \doteq u)$ *with accessor expressions* $a_i \in acc\_list$ *and a logic variable* p_j *for each* $* \in \{a_1, \ldots, a_n\}$ *of the corresponding type.*

Example 15.9. Let terms o, u denote two objects of type T and term n a positive integer. The formula $\mathtt{reach}[next;](\mathtt{o}, \mathtt{u}, \mathtt{n})$ is valid in a state s iff $s \models o\underbrace{.next. \cdots .next}_{n} \doteq u$. Let t_1, t_2 be terms of type `ASTNode` then $\mathtt{reach}[children[*];](\mathtt{t}_1, \mathtt{t}_2, 2)$ is valid in state s iff $s \models \exists p_1 \exists p_2; (t_1.children[p_1].children[p_2] \doteq t_2)$.

Furthermore $\mathtt{reach}[acc\_list;](\mathtt{o}, \mathtt{u}, 0)$ will always be equivalent to $o = u$.

The taclet reachableDefinition representing the semantics definition of the reachable predicate in its instance for $acc\_list = children[*];$ can be written as follows:

— KeY —
```
reachableDefinition {
  \find(reach[children[*];](t1, t2, n))
  \varcond(\notFreeIn(k, t1, t2, n))
  \replacewith(t1 = t2 & n = 0 |
      (t1 != null & n > 0 &
       \exists k;( k>=0 & k<t1.children@(HeapObject)).length &
        reach[children[*];*.children.length;]
           (t1.children@(HeapObject)[k], t2, n-1))) )
};
```
— KeY —

The given rule defines reachable recursively, but is well-founded. Instead of `reach`[`children`[∗];] the slightly shorter form `reach`[`children`[∗];] will from now on be used in order to keep the formulas readable.

If `n` is negative, it will evaluate to false due to `n` >= `0`. If `n` is equal to zero then `t1` = `t2` must hold. This is the only case where `t1` may be **`null`**. If `n` > `0` then it must hold that `t2` is reachable in `n` − `1` steps from an object stored in the children array of `t1`.

Together with induction over the natural numbers, rule reachableDefinition suffices to express the required reachable properties. But it would not be very convenient and, therefore, a number of further taclets exists covering common situations directly

— KeY —
```
reachableDefinitionBase {
     \find(reach[children[*];](t1, t2, 0))
     \replacewith(t1 = t2)
};

reachableDefinitionFalse {
     \assumes (n < 0 ==>)
     \find(reach[children[*];](t1, t2, n)) \sameUpdateLevel
     \replacewith(false)
};
```
— KeY —

Encoding the Backtracking Path

For the specification of the loop invariant it turns out to be useful to define a relation *onPath*, which describes the backtracking path. We formalise the relation as the characteristic function of the set of nodes lying on the backtracking path in means of an auxiliary non-rigid predicate with explicit dependencies `onPath`[∗.`children`[∗]; ∗.`nextChild`] : $HeapObject \times HeapObject \times int$.

For sake of shortness and readability, we will skip the accessor list for the *onPath* predicate from now on. Formally, the non-rigid predicate *onPath* is defined as follows: Let σ denote a state, x, y terms of type `HeapObject` and n an integer term, then:

$$\sigma \models onPath(x, y, n)$$

iff.

$n >= 0$ and there exist terms $x = u_0, \ldots, u_n = y,$ such that for all $0 \leq i < n$:

$$\sigma \models u_{i+1} \doteq u_i.children[u_i.nextChild - 1]$$

and

$$\sigma \models ! \, u_i \doteq null$$

Notice that $onPath(x, y, 0)$ is equivalent to $x \doteq y$. The semantical definition of *onPath* is reflected by the calculus in form of a recursive definition:

— KeY —

```
onPathDefinition {
  \find(onPath(t1,t2,step))
  \replacewith(
    step >= 0 &
    ((t1 = t2 & step = 0) |
     (t1 != null & t1.nextChild@(HeapObject) > 0 &
      t1.nextChild@(HeapObject) <
             t1.children@(HeapObject).length &
      onPath(t1.children@(HeapObject)
                  [t1.nextChild@(HeapObject)-1], t2, step-1))
    ))
 };
```

— KeY —

The recursion is well founded and required to formalise the existential statement given in the semantical definition. For convenience reasons, we use additional taclets which can be derived directly from the rule onPathDefinition:

— KeY —

```
  onPathBase {
        \find ( onPath(t1, t2, 0) )
        \replacewith( t1 = t2 )
  };
  onPathNull {
        \find ( onPath(null, t2, n) )
        \replacewith( n = 0 & t2 = null )
  };
```

```
onPathNegative {
      \assumes (n < 0 ==> )
      \find ( onPath(t1, t2, n) )  \sameUpdateLevel
      \replacewith( false )
};
```

KeY

With this work done, we can now express the property that a node `x` is on the backtracking path by:

$$\backslash\text{exists } int\ n\ onPath(previous, x, n); \ \mid \ \mathtt{x} = \mathtt{current}$$

where `previous` and `current` are the reference variables declared in method `mark`. Note, that we have included the `current` node to belong to the backtracking path.

15.2.2 Specification in Java Card DL

Pre- and Postconditions

The proof obligation of method `mark` to be proven valid is listed in Fig. 15.4. In previous chapters we considered proof obligations in OCL or JML. Since the reachable concepts are available in neither of them we resort to using Dynamic Logic formulas directly (⇒ Chap. 14). The proof obligation is composed of three components:

1. invariant of class `HeapObject` (lines 1–12),
2. the precondition proper (lines 14–19) and
3. the postcondition (lines 22–26) to be ensured to hold after the method has been executed.

The instance invariant of class `HeapObject` gives the following guarantees:

Line 3 that field `children` is always a non null array reference. Consequently, a node representing a sink refers to a zero-length array instead to **`null`**.
Lines 4–5 that the value of field `nextChild` ranges from 0 to the number of children.
Lines 7–9 that arrays referenced by `children` are not shared among different `HeapObject` instances.
Lines 10–12 that the components of the `children` array are not null.

In addition, a caller of method `mark` has to ensure that the start node, which is passed through as an argument (`startNode`), is not **`null`** (line 15) and that all markers of all nodes have been reset to their initial values indicating that they have not yet been visited (line 16). We simplified the specification

KeY

```
   // Invariant of class HeapObject
 2   \forall HeapObject ho;(!(ho = null) ->
        !(ho.children = null)              &
 4      ho.nextChild >= 0                  &
        ho.nextChild <= ho.children.length &
 6      ho.children.length >= 0 ) &
     \forall HeapObject ho1;
 8      \forall HeapObject ho2;(!(ho1 = ho2) ->
           !(ho1.children = ho2.children)) &
10   \forall HeapObject ho;\forall int i;
          (!(ho = null) & 0 <= i & i < ho.children.length ->
12             !(ho.children[i] = null))
   // contract for method 'mark'
14 // precondition
    !(startNode = null) &
16  \forall HeapObject ho; (ho.visited = FALSE & ho.nextChild = 0)
   // keep old values
18  \forall HeapObject ho; \forall int i;
        (children_pre(ho,i) = ho.children[i])
20 ->
    \[{ sw.mark(startNode); }\]
22 // postcondition
      (\forall HeapObject ho;(ho != null &
24        \exists int n; (n>=0 &
             reach[children[*];](startNode, ho, n)) <->
26           ho.visited = TRUE))
```

KeY

Fig. 15.4. The Java Card DL proof obligation for verifying Schorr-Waite

slightly by requiring that the markers of *all* nodes even of not yet created ones have been set to their initial value.

By proving the proof obligation, we can ensure that all and *only* nodes reachable from the starting node `startNode` have been marked as `visited` (line 22-26). Note, that the postcondition does not specify reservedness of the class invariants. This makes the proof easier and, in fact, one would often decompose these kind of proof obligations in order to keep a proof feasible.

For later use it will be convenient to refer to the "old" content of the `children` arrays. As Java Card DL is not a high-level specification language, there is no construct like `@pre` in OCL or `\old` in JML. Instead we use in line 19 the trick to remember old values of the OCL/DL translation as described in Sect. 5.2.

Invariants

The most critical part of the specification is the loop invariant as most of the later verification depends on a sufficiently strong invariant. The **while** invariant rule used in KeY (⇒ Sect. 3.7.1) takes change information into consideration and allows to reduce the complexity of the invariant.

The loop's assignable set is

```
{current, previous, next, old,
*.children@(HeapObject)[*],
*.visited@(HeapObject),
*.nextChild@(HeapObject) }
```

In the first line all possibly altered method-local pointers are enumerated. The remaining lines denote all the fields of nodes that are likely to be changed. The assignable set is a conservative approximation in principal it would be sufficient to restrict to fields of nodes reachable from the starting node.

The loop invariant is listed in Fig. 15.5. Its core part on which we will concentrate on is the subformula in lines 20–40. We come later back to the filtering condition stated in the lines 20–25. For the next few paragraphs, assume that the first condition (lines 21– 22), and consequently, the following equations `lCur=current`, `lPrv=previous` and `bnd=current.nextChild` hold.

To write a good invariant means to find the right balance between being strong enough to allow to prove the methods postconditions, but not too strong in order to keep the *preserves loop invariant* proof branch as simple as possible.

For the moment let the graph to be marked be similar to the one shown in Fig. 15.6. In both sub-figures the algorithm is currently at node c and prepares for its move onwards to node n. The other children $d_0 \dots d_k$ (with $k = c.\texttt{nextChild}-1$) have already been accessed via node c. The backtracking path is highlighted using solid curved edges.

In order to get a rough idea, how a possible invariant could look like, we concentrate first on the case illustrated by the left part of Figure 15.6(a).

The subgraph S spanned by the current node's children d_0 to d_k is a promising candidate to look at for a loop invariant. One is tempted to state that all nodes belonging to S have already been marked as visited and in fact, that is what we express in lines 27–31. The proof plan in mind is that if this invariant is preserved by the loop, then when the loop terminates we are back at the start node *and* all of its children have been accessed. Thus we can yield directly from the loop invariant that all nodes in the subgraph spanned by its children, which is nearly the complete graph excluding just the start node itself, have been marked.

But unfortunately the proposed invariant is not preserved by the loop, due to situations like the one illustrated in Fig. 15.6(b). In such a scenario the spanned subgraph contains a node u', which is only reachable via paths crossing the backtracking path, i.e., all paths share at least one node (here: u) with the backtracking path. In this case we cannot assume, that the complete

KeY

```
  current != null
& current.visited = TRUE
& (previous  = null -> startNode = current)
& (previous != null -> previous.nextChild > 0)
& \forall HeapObject ho;(
     (\exists int n; onPath(previous, ho, n)) ->
         (ho = null | ho.visited = TRUE))
& \forall HeapObject ho;
     (ho != null ->
        ho.nextChild <= ho.children.length & ho.nextChild >= 0)
& \forall HeapObject ho;\forall int i;
     (ho != null & 0 <= i & i<ho.nextChild ->
        (ho.children[i] != null |
         (ho = startNode & current != startNode &
            i = ho.nextChild - 1))) & ...
& \forall HeapObject ho;
     \forall int i; ((ho != null & i >= ho.nextChild & i >= 0 &
        i < ho.children.length) ->
          ho.children[i] = children_pre(ho,i))
& \forall HeapObject lCur;\forall HeapObject lPrv;\forall int bnd;
  ((lCur != null & ( (lCur = current & lPrv = previous
                   & bnd = current.nextChild)
                  | (lPrv = lCur.children[lCur.next-1]
                    & \exists int d; onPath(lPrv, lCur, d)
                    & bnd = current.nextChild - 1)))
  ->
   \forall HeapObject ho1;(
     \forall int n; (ho1 != null & n >= 0 &
       \exists int idx; (0 <= idx & idx < bnd &
         reach[children[*];](lCur.children[idx], ho1, n)))
     -> (ho1.visited = TRUE |
          (\exists HeapObject ho2; (ho2 != null &
            ho2.children != null
          & (\exists int d; (d >= 0 & onPath(lPrv, ho2, d))
               | ho2 = lCur)
          & \exists int j; (ho2.nextChild <= j
              & j < ho2.children.length
              & \exists int l; (l >= 0
                  reach[children[*];](ho2.children[j], ho1, l))
    ))))))

  Assignable Clause
  { current, previous, next, old, *.children@(HeapObject)[*],
    *.visited@(HeapObject), *.nextChild@(HeapObject) }
```

KeY

Fig. 15.5. Loop invariant and assignable clause

subgraph reachable from u has been visited. As the algorithm as described in 15.1.1 will stop at u and perform a backtracking step instead of exploring its unvisited children. Literally spoken, the backtracking path plays the role of demarcation line that bounds subgraph S.

To cope with these kinds of situations the stated property has to be weakened. This is achieved by introducing a disjunction (lines 33–40) stating now that all nodes of the spanned subgraph have been visited *or* there is at least one path to the node crossing the backtracking path. Notice, that this property is weaker than necessary as we do not require a node to be visited, if there is at least one path sharing a node with the backtracking path. This weakening does not hurt us, as in the use case the backtracking path is empty, turning the second part of the disjunction to false and allowing us to draw the conclusion that all nodes of the subgraph spanned by the children of the starting node have been marked as visited.

In order to reestablish the invariant in case of backward step, one has to state the explained property not only relative to the currently examined node (i.e., `lCur=current`), but for the other nodes on the backtracking path too. Therefore the second part of the filtering condition (lines 23– 25) is required. Some further (technical) invariant details:

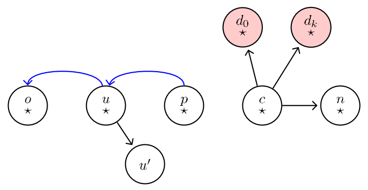
lines 5–7 in addition to the nodes specified by the invariant's core part as marked, also the nodes of the backtracking path have been marked. This property is essential
- to solve some aliasing problems occurring during the proof like that the `current` node does not coincide with the former `previous` node
- to reason that the complete graph has been visited. Remember the core part allows only to draw the conclusion that the subgraph spanned by the children has been marked visited, but excludes the `current` node

lines 9–16 express a kind of "preserve instance invariant" statement for class `HeapObject`. Note that the loop will violate the invariant that **`null`** is not referenced by the children array components. The weakened version of this invariant can be found in lines 13-16.

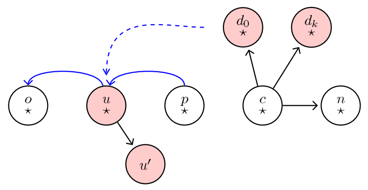
lines 16–19 completes the former part of the invariant by stating that the components of the children with an index greater or equal to `nextChild` remain unchanged, allowing to use parts of the `HeapObject`'s invariant stated in the precondition (e.g., that the stored values are not **`null`**).

15.3 Verification of Schorr-Waite Within KeY

In the subsequent sections we will roughly outline the correctness proof of Schorr-Waite. We will step only into the technical details for some of the more interesting proof steps. The interested reader may download the complete proof from the book website and load it with the accompanying KeY version.



(a) Subgraph spanned by children 0 to $nextChild - 1$ do not contain a node only reachable by through nodes on the backtracking path



(b) Subgraph spanned by children 0 to $nextChild - 1$ contains a node where all paths have to cross the backtracking path

Fig. 15.6. Loop invariant: core part

15.3.1 Replacing Arguments of Non-rigid Functions Behind Updates

In several branches of the proof, we face situations similar to the following:

——— KeY ———
```
  {current:=startNode} reach[..](current, x, n)
==>
  {current:=startNode} reach[..](startNode, x, n)
```
——— KeY ———

The sequent is clearly universally valid. But in order to close this proof goal, the first arguments of both occurrences of the reachable predicate need to be unified. Inserting the reachable definition will not succeed, as the definition itself is recursive and the value of `n` unknown. Furthermore, we have to operate behind updates, restricting the kind of applicable taclets.

In order to close this sequent the non-rigid arguments of the formulas `reach[..](fst, snd, thrd)` have to be replaced by new rigid constant symbols `c`i and defining equations `{current:=startNode}c`i `= fst` have to be added to the sequent's antecedent.

The replacement is performed by successive application of rule pullOut on the first arguments of both sides:

——— KeY ———
```
 pullOut { \find ( t ) \sameUpdateLevel
           \varcond ( \new(sk, \dependingOn(t)) )
           \replacewith (sk)
           \add ( t = sk ==>)
 };
```
——— KeY ———

The result is the following sequent:

```
— KeY ————————————————————————————————————
  {current:=startNode} (c1 = current),
  {current:=startNode} (c2 = startNode),
  {current:=startNode} reach[..](c1, x, n)
==>
  {current:=startNode} reach[..](c2, x, n)
———————————————————————————————————— KeY —
```

After a few further simplification steps, we obtain:

```
— KeY ————————————————————————————————————
  c1 = startNode, c1 = c2,
  {current:=startNode} reach[..](c1, x, n))
==>
  {current:=startNode} reach[..](c2, x, n))
———————————————————————————————————— KeY —
```

The equation `c1 = c2` contains only rigid elements and is thus applicable also in the scope of updates - in fact behind any modality.

Applying this equation on the first argument of the third formula in the antecedent `{current:=startNode} reach[..](c1, x, n))` establishes an axiom where two equal formulas occur in the ante- and succedent.

15.3.2 The Proof

Invariant Initially Valid

This branch closes almost automatically (with help from Simplify for some universal quantifier instantiations). Only one interactive step remains for the invariant part, that ensures that all nodes on the backtracking path have been marked visited (lines 5-7). In the initial case, only the starting node, which has been marked visited in the statement before the loop is entered, is part of the backtracking path. To show that no other node is on the backtracking path, we insert the *onPath* predicate definition and leave the remaining steps for the strategies.

Use Case

As the method to verify ends when the loop terminates, this proof branch is of normal complexity. Most of the steps are performed automatically by the strategies. Nevertheless some interaction with the prover are necessary. Besides usual universal quantifications, which would be possible to perform also automatically (i.e., a heuristic approach should succeed), there is one step that will reoccur in the *preserves loop invariant*, which is of particular interest.

In this branch only normal termination of the loop is considered. Abrupt termination as uncaught exceptions or `return` or `break` statements are treated in the preserves invariant branch. The task is to prove that when the

- loop condition evaluates to `false` and
- loop invariant is valid

then

- the method's postcondition is satisfied, i.e., all reachable nodes have been visited.

The plan is to use the core part of the invariant (Fig. 15.5), lines 20–40).

The postcondition to prove looked like

—— KeY ——
```
\forall HeapObject x; \forall int n;
  (x != null & reach[children[*];](startNode, x, n)
    -> x.visited = TRUE)
```
—— KeY ——

In order to prove the post condition we have to show that an arbitrary chosen non-null instance `x_0` of type `HeapObject` reachable from the starting node `startNode` within `n_0` steps, is marked reachable.

After some steps, this part of the proof goal is presented[1] as

—— KeY ——
```
( !(x_0 = null) & n_0 >= 0 &
   n_0 <= -1 + startNode.children.length &
{\for HeapObject h; h.nextChild := anonNextChild(h) ||
 startNode.visited              := TRUE             ||
 \for HeapObject h; h.visited   := anonVisited(h)   ||
 current                        := startNode        ||
 previous                       := anonPrevious(sw) ||
 \for (int i; HeapObject h)
   \if (i >= 0 & i <= -1 + h.children.length)
        h.children[i]           := anonChildren(h.children, i)}
 reach[children[*];](startNode, x_0, n_0)) ->
     anonVisited(x_0) = TRUE
```
—— KeY ——

The first line corresponds with the afore stated side conditions. Following is a quantified update describing the state after leaving the methods. The functions symbols `anon`$*$ are the anonymous functions introduced by the while invariant rule application. They describe the value of the location after the while loop, for example, `anonVisited(h)` is the value of `h.visited` when leaving the loop and so on.

[1] Names are slightly beautified.

Preserves Loop Invariant

Although this proof branch requires most interactions, the necessary techniques have been already introduced in the preceding paragraphs. The greatest difficulty is to keep track of the current loop invariant part that has to be proven.

In several subgoals the definitions of the *reachable* and *onPath* predicate have to be inserted—often in combination with the pullOut taclet to make the predicates' arguments rigid, as described in the *Use Case* paragraph. The remaining steps have been mostly simple quantifier instantiations.

15.4 Related Work

There is a variety of literature available about verification of the Schorr-Waite algorithm. We briefly describe a (representative) selection of them.

Broy and Pepper

The Schorr-Waite algorithm has been treated by [Broy and Pepper, 1982]. In this paper, the authors start with the construction of an algebraical data type modelling a binary graph. They continue with the definition of the reflexive and transitive closure relation R^* of the graph. Then a function B is developed, *proven* to compute the set of all reachable graph nodes from a distinguished node x, i.e. $R^*(x)$. The function B turns out to realise the well-known depth-first traversal algorithm for (binary) graphs.

An extended graph structure is build upon the binary graph data type. In addition to the binary graph it provides two distinguished nodes (representing the current and previous node). Also two additional basic functions *ex* and *rot* are defined, which exchange the current and previous node resp. perform a rotation operation (forward step). By composition of these elementary graph operations a function is constructed that computes and returns a tuple consisting of a set of nodes and an extended graph structure. It is proven that the returned node set is the same as computed by the former function B and that the returned extended graph structure is the same on which the function has operated.

Afterwards the functional algorithm is refined to a procedural version.

Mehta and Nipkow

The authors of [Mehta and Nipkow, 2003] verify the correctness of a Schorr-Waite implementation (for binary graphs) using higher order logics. The program is written in a simple imperative programming language designed by the authors themselves. The operational semantics of the programming language has been modelled in Isabelle/HOL and a Hoare style calculus has been derived from the semantics.

The main difference to our approach is the explicit modelling of heaps and the distinction between addresses and references. On top of these definitions a reachability relation (and some auxiliary relations) is defined as above.

The program is then specified using Hoare logic by annotating the program with assertions and a loop invariant making use of the former defined relations. From these annotations, verification conditions are generated, which have to be proven by Isabelle/HOL.

Abrial

The approach described in [Abrial, 2003] uses the B language and methodology to construct a correct implementation of the Schorr-Waite algorithm. Therefore the author starts with a high-level mathematical abstraction in B of a graph marking algorithm and then successively refines the abstraction towards an implementation of an (improved) version of Schorr-Waite. Each refinement step is accompanied by several proof obligations that need to be proven to ensure the correctness of the refinement step.

Yang

In [Yang, 2001] the author uses a relatively new kind of logic called Separation Logics, which is a variant of bunched implication logics. For verification they use a Hoare like calculus. The advantage of this logic is the possibility to express that two heaps are distinct and in particular the existence/possibility of a frame introduction rule. In short, the frame introduction rule allows to embed a property shown for a local memory area in a global context with other memory cells.

The frame rule allows to show that if $\{P\}C\{Q\}$ is valid for a local piece of code C then one can embed this knowledge in a broader context $\{P * H\}C\{Q * H\}$ as long as the part of the heap H talks about is not altered by C (separate heaps). Without this frame rule one would have to consider H when proving $\{P\}C\{Q\}$, which makes correctness proves very tedious, in particular when the property shall be used in different *separate* contexts H_i.

Hubert and Marché

In [Hubert and Marché, 2005] the authors follow an approach very similar to the one presented in this chapter. They used a weakest precondition calculus for C implemented in the CADUCEUS tool to verify a C implementation of Schorr-Waite working on a bigraph. In the same manner as described here, they specified the loop invariant with help of an inductively defined reachable predicate using a higher order logic.