Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 259 (2009) 143–163
www.elsevier.com/locate/entcs

Data Refinement of Invariant Based Programs

Viorel Preoteasa[1] and Ralph-Johan Back[2]

Department of Information Technologies
Åbo Akademi University
Turku, Finland

Abstract

Invariant based programming is an approach where we start to construct a program by first identifying the basic situations (pre- and postconditions as well as invariants) that could arise during the execution of the algorithm. These situations are identified before any code is written. After that, we identify the transitions between the situations, which will give us the flow of control in the program. Data refinement is a technique of building correct programs working on concrete data structures as refinements of more abstract programs working on abstract data types. We study in this paper data refinement for invariant based programs and we apply it the the construction of the classical Deutsch-Schorr-Waite graph marking algorithm. Our results are formalized and mechanically proved in the Isabelle/HOL theorem prover.

Keywords: Invariant based programming, Data refinement, Mechanical verification

1 Introduction

Invariant based programming [3,4,5,8] is an approach to constructing correct programs where we start by identifying all basic situations (pre- and post-conditions, loop invariants, etc.) that could arise during the execution of the algorithm. These situations are determined and described before any code is written. After that, we identify the transitions between the situations, which together determine the flow of control in the program. The transitions are verified at the same time as they are constructed. The correctness of the program is thus established as part of the construction process.

We use a diagrammatic approach to describe invariant based programs, *(nested) invariant diagrams*, where situations are shown as (possibly nested) boxes and transitions are arrows between these boxes. We associate a collection of constraints with each situation box, and a sequence of simple statements with each transition arrow. Nesting expresses the information content of the situations: if situation B is nested

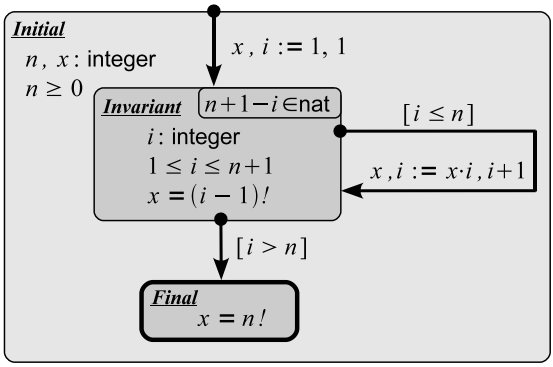
[1] Email: viorel.preoteasa@abo.fi
[2] Email: backrj@abo.fi

doi:10.1016/j.entcs.2009.12.022

within situation A, then B inherits the constraints of A. The control structure is secondary to the situation structure, and will usually not be well-structured in the classical sense, i.e., control is not necessarily expressible in terms of single- entry single-exit constructs. The invariant diagram shows explicitly all the information needed in order to verify that the program is correct: pre- and post-conditions, invariants, transitions, and termination functions.

We have been experimenting with teaching formal methods using invariant based programming for a number of years now, mainly in the form of introductory CS courses at university levels. The experiences have been good, the students learn quite easily how to construct programs that are correct by construction with this approach, and appreciate the added understanding that the approach brings to how a program works. The problems encountered have less to do with the invariants first approach than with the general problem of how to describe formally (in predicate calculus) situations that are intuitively well understood [5].

The following shows a simple example of an invariant based program, the factorial function, expressed as an invariant diagram.



There are three situations here, *Initial, Invariant,* and *Final.* The initial situation declares the program variables n and x and restricts them to range over integers. In addition, we require that $n \geq 0$. The two other situations are nested inside *Invariant,* which means that they inherit the program variables and constraints from the outer situation. Situation *Final* states that upon termination $x = n!$ must hold. The intermediate situation *Invariant* declares an additional program variable i and restricts it to range over integers in the range 1 to $n + 1$. In addition, it requires that $x = (i - 1)!$ holds in this situation. There are three transitions in the diagram, one leading from *Initial* to *Invariant* providing initial values for x and i, one leading from *Invariant* back to itself that updates the variables x and i when $i \leq n$ holds, and one leading from *Invariant* to *Final* that is taken when $i > n$ holds but which does not change any program variables.

The *Invariant* situation also gives a termination function, in the form *termination function* $\in W$, where W is some well founded set. In this case, the well founded set is the set of natural numbers *nat* and the termination function is $n + 1 - i$. The property $n+1-i \in nat$ must be provable from the constraints that hold in situation *Invariant* (in this case, this amounts to proving that $0 \leq n + 1 - i$ holds).

Execution of an invariant based program may start in any situation (not necessarily an initial situation), in a state that satisfies the constraints of the situation. One of the transitions that are enabled in this situation will be chosen. The transition is then executed, leading to new state in (possibly) another situation. There again one of the enabled transitions is chosen, and executed, and so on. In this way, execution proceeds from situation to situation. Execution terminates when a situation (not necessarily a final situation) is reached in a state for which there are no enabled transitions. Because the execution could start and terminate in any situation, invariant-based programs can be thought of as multiple entry, multiple exit programs. We may choose to identify some situations in an invariant based program as *initial situations* and some other situations as *final situation,* with the idea that execution should start in some initial situation and it should end in some final situation.

An invariant based program is *consistent*, if each transition *preserves* the situation constraints. This means that if we start execution in a situation A and in a state where the constraints of A are satisfied, and choose a transition that is enabled in A, then executing the transition will lead to some situation B (which could be A again) such that the resulting state satisfies the constraints associated with B. An invariant based program is *terminating*, if each execution of the program eventually terminates. For a given collection of final situations, an invariant based program is said to be *live* if termination only occurs in some final situation. The semantics and proof theory for invariant based programs are studied in detail in [8].

The purpose of this paper is to study the use of *data refinement* when building invariant based programs. Data refinement [12,2,10,11] is a technique of building correct programs working on concrete data structures as refinements of more abstract programs working on abstract data structures. The correctness of the final program follows from the correctness of the abstract program and from the correctness of the data refinement. The overall complexity of the correctness proof is usually lower when using data refinement than when the final program is developed directly on the concrete data structure. We will show how to adapt the basic idea of data refinement to the construction of invariant based programs. At the same time, we will extend the notion of nested invariant diagrams in a way that makes it easy to describe data refinement. On the theoretical side, this paper extend the work described in [8] by including methods for carrying out data refinement. The data refinement theorems that we present here have all been proved mechanically in Isabelle/HOL [15].

We apply our technique to a larger case study, constructing the classical Deutsch-Schorr-Waite (DSW) [16,13] marking algorithm for arbitrary graphs. The DSW algorithm marks all nodes in a graph that are reachable from a root node. The marking is achieved using only one extra bit of memory for every node. The graph is given by two pointer functions, left and right, which for any given node return its left and right successors, respectively. While marking, the left and right functions are altered to represent a stack that describes the path from the root to the current node in the graph. On completion the original graph structure is restored. We construct the DSW algorithm by a sequence of three successive data refinement steps. The

proof obligations that arouse during our development were all formalized an proved mechanically in Isabelle/HOL. The main difference in our case study, as compared to the previous studies [14,1], is that the whole refinement process is carried out using invariant diagrams. We also believe that the way we develop the algorithm by first proving a generalization of the algorithm significantly reduces the overall proof effort.

The paper is structured as follows. Section 2 presents the proof theory for invariant based programs. Section 3 shows how to carry out data refinement of invariant diagrams. The DSW algorithm is constructed in Section 4. Section 5 presents some concluding remarks.

2 Proof theory for invariant diagrams

We describe here more precisely how to prove the correctness of invariant diagrams. We start by defining the notion of *monotonic predicate transformers,* on which the semantics of invariant based programs are based, and then continue with defining the different aspects of the correctness for invariant diagrams.

2.1 Monotonic predicate transformers

We assume a given *state space* Σ. A *state* $\sigma \in \Sigma$ is a tuple with one component for every program variable and it represents the values of the program variables at some moment of the computation. We assume that the value of a program variable x is given by a projection function $x : \Sigma \to T$ where T is the type of x.

Predicates (or *sets*), denoted pred, are functions from $\Sigma \to \mathsf{bool}$. We denote by $\cup$, $\cap$, $\subseteq$ the *union, intersection*, and *inclusion* of predicates respectively.

We write mtran for the type of all *monotonic predicate transformers*, i.e., monotonic functions from $\mathsf{pred} \to \mathsf{pred}$. *Programs* are modeled as elements of mtran. If $S \in \mathsf{mtran}$ and $p \in \mathsf{pred}$, then $S.p \in \mathsf{pred}$ are all states from which the execution of S terminates in a state satisfying the post-condition p. *Sequential composition* of programs, denoted $S\,;\,T$, is defined as the functional composition of monotonic predicate transformers, i.e. $(S\,;\,T).p = S.(T.p)$. We denote by $\sqsubseteq$, $\sqcup$, and $\sqcap$ the point-wise extension of $\subseteq$, $\cup$, and $\cap$, respectively. The type mtran, together with the point-wise extension of the operations on predicates, forms a complete lattice. The partial order $\sqsubseteq$ on mtran is the *refinement relation* [9,2]. The predicate transformer $S \sqcap T$ models *demonic choice* - the choice between executing S or T is arbitrary and cannot be influenced from outside.

Program expressions of some type T are seen as functions from $\Sigma \to T$. If $e : \Sigma \to T$ and $\sigma \in \Sigma$, then $e.\sigma$ is the value of expression e in state σ, where the values of the free variables in e are given by σ. For example the value of expression $x + y$ in a state σ where $x.\sigma = 3$ and $y.\sigma = 4$ is 7. In Isabelle/HOL we implement a program expression (predicate) e which contains the free variables $x_i : T_i$ as a function $e : T_1 \to \ldots \to T_n \to T$. Then $e.a_1 \ldots . a_n$ is the value of the expression e in the state σ where $x_i.\sigma = a_i$. If e, f are program expressions and x is a program variable, then we denote by $e[x := f]$ the substitution of x with f in e.

We introduce a number of basic monotonic predicate transformers that we use in constructing invariant based programs. These are the *assignment statement* $x := e$, the *assume statement* $[p]$, the *demonic assignment statement* $[x := x' \mid b]$, the *angelic assignment statement* $\{x := x' \mid b\}$ and the *magic statement* magic. We define these as follows:

Assignment	$(x := e).q = q[x := e]$
Assume	$[\,p\,].q = \neg p \cup q$
Demonic assignement	$[\,x := x' \mid b\,].q = (\forall x' \bullet b \Rightarrow q[x := x'])$
Angelic assignement	$\{x := x' \mid b\}.q = (\exists x' \bullet b \wedge q[x := x'])$
Magic	$\mathsf{magic}.q = \mathsf{true}$

The variable x' is bound in the demonic and angelic assignment statements, and we assume that q does not contain the variable x' free.

All the above program constructs except the angelic assignment can be reduced to a demonic assignment. Moreover, sequential composition and demonic choice of demonic assignments are also a demonic assignments. Because of these properties we will always work with demonic assignments in definitions, but we will use all of the above program constructs in examples. We could also allow *assert statements*, but the treatment becomes a little bit simpler when we omit it (assert statements can be handled without problems, as shown in [8]).

For a monotonic predicate transformer $S \in \mathsf{mtran}$ we introduce the *guard* of S by $\mathsf{grd}.S = \neg(S.\mathsf{false})$.

Theorem 2.1 *The following properties are true.*

$\mathsf{grd}.(x := e) = \mathsf{true}$
$\mathsf{grd}.[\,p\,] = p$
$\mathsf{grd}.[\,x := x' \mid b\,] = (\exists x' \bullet b)$
$\mathsf{grd}.(S_1 \sqcap S_2) = \mathsf{grd}.S_1 \cup \mathsf{grd}.S_2$

If p and q are predicates and S is a program, then a *total correctness triple*, denoted $p \,\{\!|\, S \,|\!\}\, q$, is true if and only if $p \subseteq S.q$.

For a function $f : X \to Y$ we denote by $f[x := y] : X \to Y$ the function given by

$$f[x := y].z = \begin{cases} y & \text{if } x = z \\ f.z & \text{otherwise} \end{cases}$$

2.2 Invariant diagrams

An *invariant diagram* is a directed graph where nodes are labeled with *invariants* (predicates) and edges are labeled with *program statements* (monotonic predicate transformers). The nodes of the invariant diagram are called *situations* and the edges are called *transitions*.

Let I be a nonempty set if indexes. Formally, an invariant diagram is a tuple (P, T) where $P : I \to \mathsf{pred}$ are the *invariants* and $T : I \times I \to \mathsf{mtran}$ are the *transitions*. We also call T a *transition matrix*. The guard of a transition matrix in a situation $i \in I$, $\mathsf{grd}.T.i \in \mathsf{pred}$ is the disjunction of the guards of all transitions from i:

$$\mathsf{grd}.T.i = \bigvee_{j \in I} \mathsf{grd}.T.i.j$$

We assume in this paper that edges are labeled only with demonic assignment statements $[x := x' \bullet R.x.x']$. In general, we can have any statements constructed using sequential composition of assume, assignment, and demonic assignment statements on the edges. However, such statements can all be reduced mechanically to a single equivalent demonic assignment statement.

For example, the program for computing the factorial of a number n is represented as a invariant diagram by choosing $I = \{\mathit{Initial}, \mathit{Invariant}, \mathit{Final}\}$, and giving values to T according to the diagram. A transition from situation i to situation j is assumed to be magic (the transition which is never enabled) if it is not drawn explicitly in the diagram. Thus, for instance,

$$T(\mathit{Invariant}, \mathit{Invariant}) = [x \leq i] \,;\, x, i := x \cdot i, i + 1$$

and

$$T(\mathit{Invariant}, \mathit{Initial}) = \mathsf{magic}.$$

The execution of an invariant diagram may start in any situation $i \in I$, and then non-deterministically choose some enabled transition from i leading to some new situation $j \in I$. The execution continues in this way as long as transitions are enabled. The execution terminates when a situation i is reached where no transitions are enabled. In [8], we have introduced operational semantics and predicate transformer semantics for invariant based programs and we proved their equivalence. We have also introduced correct and complete proof rules for invariant diagrams. We recall these proof rules below.

Informally, a transition matrix $T : I \times I \to \mathsf{mtran}$ is totally correct with respect to the initial predicates $P : I \to \mathsf{pred}$ and final predicates $Q : I \to \mathsf{pred}$, denoted $\vdash P \,\{\!|\, T \,|\!\}\, Q$, if for all initial states s and situations i for which $P.i.s$ is true, the execution always terminates, and $Q.j.s'$ is true for the termination state s' and termination situation j.

We have the following general rule for proving total correctness of invariant diagrams. If $(W, <)$ is a well founded set and $X_w : I \to \mathsf{pred}$, $w \in W$, is a family of predicates, then the following inference rule is true:

$$\frac{((\vee X) \subseteq P) \wedge ((\vee X) \cap \neg \mathsf{grd}.T \subseteq Q) \wedge (\forall i, j, w \bullet \vdash X_w.i \,\{\!|\, T_{i,j} \,|\!\}\, X_{<w}.j)}{\vdash P \,\{\!|\, T \,|\!\}\, Q} \tag{1}$$

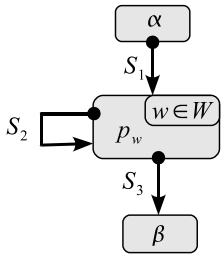
where

$$X_{<w} = \bigvee_{v<w} X_v \quad \text{and} \quad \vee X = \bigvee_{w \in W} X_w$$

An invariant diagram (P, T) is correct if $\vdash P \{\!|\, T \,|\!\} P$.

Rule (1) requires proving that a variant decreases on every transition. In [8] we have introduced a version of this rule which only requires proving that a variant decreases in some transitions which are part of cycles in the diagram. However, as our main example has the simple structure of the diagram (2), we will not discuss further the improved rule for proving total correctness of invariant diagrams, but be content with describing how one proves correctness of invariant diagrams of this simple form.

We will in the sequel focus on diagrams of the following simple form:



(2)

We write here $w \in W$ in the upper right corner of the loop invariant, to indicate that loop invariant is in fact $\vee p = \bigvee_{w \in W} p_w$, even if we just write p_w as the situation constraint.

The proof obligations for this diagram are

$$\alpha \{\!|\, S_1 \,|\!\} \vee p$$

$$p_w \{\!|\, S_2 \,|\!\} p_{<w}$$

$$\vee p \{\!|\, S_3 \,|\!\} \beta$$

where $\vee p$ and $p_{<w}$ are defined similarly to $\vee X$ and $X_{<w}$.

In practical examples, the predicate p_w in diagram 2 is the conjunction of the situation invariant P and a formula of the form $t_i = w$, where $w \in W$ and t_i is a variant term ranging over the set W. In the invariant diagram, we then state the variant t_i by writing $t_i \in W$ in the upper right corner of situation i (as exemplified in the factorial example).

The proof obligations for the factorial example above using this more practical proof rule are as follows:

$$n \geq 0 \{\!|\, x, i := 1, 1 \,|\!\} n \geq 0 \wedge x = (i-1)! \wedge i \leq n+1$$

$$n \geq 0 \wedge x = (i-1)! \wedge i \leq n+1 \wedge n - i + 1 = w \{\!|\, [i \leq n] \,;\, x, i := x \cdot i, i+1 \,|\!\}$$
$$n \geq 0 \wedge x = (i-1)! \wedge i \leq n+1 \wedge n - i + 1 < w$$

$$n \geq 0 \wedge x = (i-1)! \wedge i \leq n+1 \{\!|\, [i > n] \,|\!\} n \geq 0 \wedge x = n!$$

The execution of a diagrams can start from any situation, and it can end in any situation. In practice, we are mainly interested in diagrams in which the execution

is guaranteed to terminate in some predetermined *final* situations. For example we want the factorial diagram presented earlier to always terminate in situation *Final*. An *invariant diagram with final situations* is a tuple (P, T, J) where (P, T) is an invariant diagram, and $J \subseteq I$ is a non-empty set of final situations. The diagram (P, T, J) is correct if (P, T) always terminates in a situation in J (we then say that the diagram is *live* for the final situations J). Formally we define the total correctness of an invariant diagram with final situations by the triple

$$\vdash P \,\{\!|\, T \,|\!\}\, P^J \tag{3}$$

where for all $i \in I$

$$P^J.i = \begin{cases} P.i & \text{if } i \in J \\ \mathsf{false} & \text{otherwise} \end{cases}$$

If we require termination only in situation *Final* for the factorial example, then we have two additional proof obligations (which are both trivial to establish):

$$n \geq 0 \Rightarrow \mathsf{true}$$

$$n \geq 0 \wedge x = (i-1)! \wedge i \leq n+1 \Rightarrow (i \leq n \vee i > n)$$

We emphasize the final situations on the diagram by drawing them with a ticker border line as already shown in the factorial example.

An *executable invariant diagram* is one in which all statements in the diagram are equivalent to the form $[\,p\,]\,;\, x := e$, where e and p are ordinary program expressions. They should thus not contain quantifiers or specification functions which are not part of the target programming language.

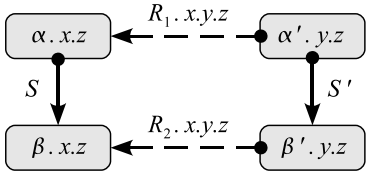
Very often in examples it is convenient to draw more than one transition between two situations i and j. We interpret these as standing for a single transition that is the demonic choice of all transitions between i and j.

3 Data refinement of invariant diagrams

Data refinement has proved to be a powerful tool in developing software systems. When writing a complex program, it is often useful to start with an abstract program on an abstract data structure, and gradually refine it to a more concrete program working on a concrete data structure. Using this approach the overall proof work is split in smaller tasks [2,10,11].

3.1 Data refinement of statements

The following diagram describes data refinement:



The *abstract* statement S modifies the global variables z and abstract variables x, and leads from an initial abstract situation $\alpha.x.z$ to a final abstract situation $\beta.x.z$. The *concrete* statement S' modifies the global variables z and concrete variables y and leads from an initial concrete situation $\alpha'.y.z$ to a final concrete situation $\beta'.y.z$. The *data abstraction relation* $R_1.x.y.z$ describe how the abstract variable x is related to the concrete variables y and z in the initial situation. Similarly for $R_2.x.y.z$ in the final situation.

We can define data abstraction in terms of an angelic statement D that computes for each concrete state some abstract state. Let us define

$$D_1 = \{x := x' \mid R_1.x'.y.z \wedge \alpha'.y.z\}$$

$$D_2 = \{x := x' \mid R_2.x'.y.z \wedge \beta'.y.z\}$$

Here the abstraction (*decoding)* statements D_i include both the abstraction relation and the concrete invariant. We say that the program S is data refined by S' via D_1 and D_2, denoted $S \sqsubseteq_{D_1,D_2} S'$, if $D_1; S \sqsubseteq S'; D_2$ holds.

Let us now further assume that the abstract statement S is just a demonic assignment,

$$S = [x,\, z := x',\, z' \mid Q.x.z.x'.z']$$

We want to refine the abstract statement in the context where the initial situation $\alpha.x.z$ holds. This is expressed by the data refinement $\{\alpha.x.z\}; S \sqsubseteq_{D_1,D_2} S'$. This holds if and only if

$$\alpha'.y.z \wedge R_1.x_0.y.z \wedge \alpha.x_0.z \wedge z = z_0 \,\{\!|\, S' \,|\!\}\, (\exists x \bullet R_2.x.y.z \wedge \beta'.y.z \wedge Q.x_0.z_0.x.z)$$

3.2 Data refinement of invariant based programs

Data refinement is often used to implement a data module with information hiding. The specification of the module defines the effect of access procedures in terms of abstract variables. The implementation of the module is in fact done in term of concrete variables, in order to achieve efficiency. If we can prove data refinement for all access methods, then any user of the module will never see a difference, and may use the module and reason about its behavior as if it was really implemented in terms of the abstract variables.

The situation with invariant based programs is different. Here we are interested in deriving a concrete algorithm working on concrete variables. The abstraction is only useful if it saves us some verification effort, or can simplify the understanding and/or construction of the algorithm. However, it turns out that data refinement is quite useful for this purpose also. In many cases, it is easier to first design an abstract program, working on some abstraction of the intended state, and prove that it satisfies our requirements, and then refine this to a more concrete program that works on the state space that we really want to (or have to) use.

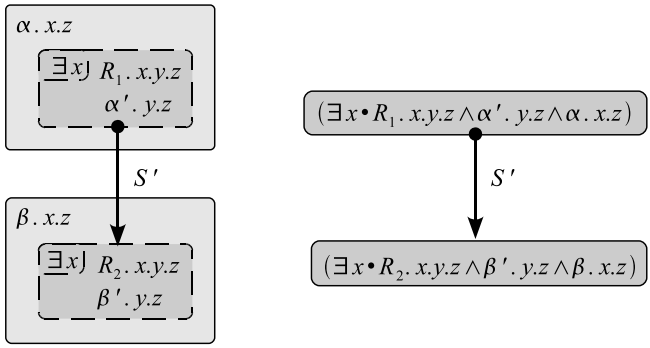
Consider the situation in diagram above. The initial situation has the constraint $\alpha'.y.z$ and the final situation has the constraint $\beta'.y.z$. Assume now that we want the concrete variables y and z to also satisfy the constraint $D_1.(\alpha.x.z)$, which says that there exists some value x such that $R_1.x.y.z \wedge \alpha.x.z$ holds. In other words, the variables y and z should also represent some abstract variables x that satisfy the abstract situation constraint. This means that the initial situation has the overall constraint

$$(\exists x \bullet R_1.x.y.z \wedge \alpha'.y.z \wedge \alpha.x.z)$$

Similarly, we can argue that the final situation has the overall constraint

$$(\exists x \bullet R_2.x.y.z \wedge \beta'.y.z \wedge \beta.x.z)$$

This divides the constraint of the situation into two parts, a concrete and an abstract part. We are now looking for a concrete transition S' that leads from the initial situation to the final situation, when interpreted in this way. We describe this in an invariant diagram as shown below on the left:



The notation on the left indicates that the invariant of the nested situation is the conjunction of the constraints in the outer situations and the inner situation, but such that the variable x is removed by existentially quantifying it. The right hand diagram is equivalent to the left hand diagram, but written without the data abstraction notation. The advantage of the left hand side notation is that it shows the structure of the situation, how it is built up from a concrete and an abstract requirement.

The transition S' is now correct if

$$D_1.(\alpha.x.z) \; \{\![\; S' \;]\!\} \; D_2.(\beta.x.y) \tag{4}$$

or, writing it out explicitly, if

$$(\exists x \bullet R_1.x.y.z \wedge \alpha'.y.z \wedge \alpha.x.z) \; \{\!| \; S' \; |\!\} \; (\exists x \bullet R_2.x.y.z \wedge \beta'.y.z \wedge \beta.x.z)$$

holds.

We prove (4) by proving two smaller steps:

(i) the abstract transition S is correct, and

(ii) the concrete transition S' is a data refinement of the abstract transition S.

This is stated in the next theorem.

Theorem 3.1

$$\alpha.x.z \; \{\!|\, S \,|\!\} \; \beta.x.z \qquad (i)$$

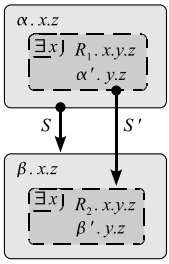
$$\wedge$$

$$\alpha'.y.z \wedge R_1.x_0.y.z \wedge \alpha.x_0.z \wedge z = z_0 \; \{\!|\, S' \,|\!\} \; \beta'.y.z \wedge (\exists x \bullet R_2.x.y.z \wedge Q.x_0.z_0.x.z) \qquad (ii)$$

$$\Rightarrow$$

$$D_1.(\alpha.x.z) \; \{\!|\, S' \,|\!\} \; D_2.(\beta.x.z)$$

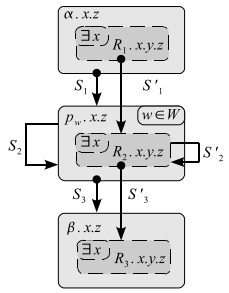
Proving assumptions (i) and (ii) in the theorem will in fact establish the correctness of the following diagram.



This shows both the abstract transition and the concrete transition, and explains how the concrete transition is derived from the abstract transition. The verification of both the abstract and the concrete transition can be done using only information that is explicitly given in the diagram.

3.3 Termination

We show next that termination of the abstract program is inherited by the concrete program. Consider the following simple diagram, with a control structure similar to the factorial function.



(5)

If $S_i = [x,\, z := x',\, z' \bullet Q_i.x.z.x'.z']$, then the proof obligations for total correct-

ness of the abstract program are

$$\alpha \{\!|\, S_1 \,|\!\} \vee p$$

$$p_w \{\!|\, S_2 \,|\!\} p_{<w}$$

$$\vee p \{\!|\, S_3 \,|\!\} \beta$$

and the proof obligation for the data refinement are

$$R_1.x_0.y.z \wedge \alpha.x_0.z \wedge z = z_0 \{\!|\, S_1' \,|\!\} (\exists x \bullet R_2.x.y.z \wedge Q_1.x_0.z_0.x.z)$$

$$R_2.x_0.y.z \wedge (\vee p).x_0.z \wedge z = z_0 \{\!|\, S_2' \,|\!\} (\exists x \bullet R_2.x.y.z \wedge Q_2.x_0.z_0.x.z)$$

$$R_2.x_0.y.z \wedge (\vee p).x_0.z \wedge z = z_0 \{\!|\, S_3' \,|\!\} (\exists x \bullet R_3.x.y.z \wedge Q_3.x_0.z_0.x.z)$$

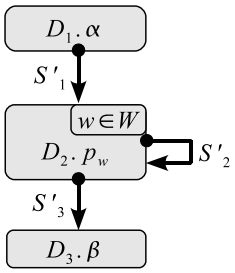
Using Theorem 3.1 we obtain

$$\begin{array}{c} D_1.(\alpha.x.z) \{\!|\, S_1' \,|\!\} D_2.(\vee p.x.z) \\ D_2.(p_w.x.z) \{\!|\, S_2' \,|\!\} D_2.(p_{<w}.x.z) \\ D_2.(\vee p.x.z) \{\!|\, S_3' \,|\!\} D_3.(\beta.x.z) \end{array} \tag{6}$$

where $D_i = \{x := x' \bullet R_i.x'.y.z\}$. Because D_i is disjunctive the relations (6) are equivalent to

$$\begin{array}{c} D_1.(\alpha.x.z) \{\!|\, S_1' \,|\!\} \bigvee_w (D_2.(p_w.x.z)) \\ D_2.(p_w.x.z) \{\!|\, S_2' \,|\!\} \bigvee_{v<w} (D_2.(p_v.x.z)) \\ \bigvee_w (D_2.(p_w.x.z)) \{\!|\, S_3' \,|\!\} D_3.(\beta.x.z) \end{array} \tag{7}$$

which ensures the total correctness of the following concrete invariant diagram.



(8)

In other words, if we can prove that the abstract program terminates, then any data refinement of the abstract program will also terminate. We have shown earlier how termination is proved for programs with the structure (5). This can be easily generalized to arbitrary diagrams because the termination argument is proved for every transition, and refinement preserves this property.

3.4 Liveness

We have seen above that termination for the abstract invariant diagram also implies termination of the concrete diagram. The same thing does not, however, hold for liveness. If we require that termination only happens in some specific final situations, then we need to prove this property explicitly for the concrete diagram. Termination in a specific final situation is thus not preserved by data refinement. (But the converse is actually true: if the concrete program is live, then the abstract program is also live.)

The simplest way of guaranteeing that an invariant based program is live is to check for each non-terminal situation that at least one transition is always enabled in any state that satisfies the situation constraints. This is usually easy to check.

4 Data refinement of the DSW marking algorithm

We will now apply the data refinement technique presented above to construct the classical Deutsch-Schorr-Waite marking algorithm as an invariant based program. This algorithm marks all reachable nodes in an arbitrary directed graph, using only one bit of extra memory for every graph node. The algorithm is given for a graph structure represented using two pointers, left and right, associated with each node. A marking bit is associated with every node, and initially the marking bit is false for all nodes.

An auxiliary bit called atom is also associated with every node (initially this bit can contain any value). The algorithm will mark exactly those nodes that are reachable from a given root node by a path on which all nodes have the atom bit false. Thus, y will not be marked if every paths from the root to y contains a node with the atom bit true. If the algorithm should mark all reachable nodes, then the atom bit must initially be false for all nodes.

The algorithm changes the left and right pointers and the atom bit while performing the marking, but will restore the original values to these variables upon completion. The original algorithm is rather asymmetric with respect to the treatment of the left and right pointers. In order to simplify the algorithm we generalize it to solve the marking problem for a graph structure in which each node has a collection of pointers $lnk.i$ for $i \in I$, instead of left and right only. This change enables us to treat uniformly the $lnk.i$ pointers (in the original algorithm we would need pairs of lemmas for both left and right pointers, however in the generalized version all these are replaced by single lemmas about the $lnk.i$ pointers).

We construct the classical DSW marking algorithm in four steps. First we build an algorithm that marks all nodes that can be reached from a special root node in an arbitrary directed graph. The graph structure is given by a relation $\mathsf{next} \subseteq \mathsf{node} \times \mathsf{node}$. This initial algorithm uses an auxiliary variable X ranging over sets of nodes. The set X is initialized with the root element and, as long as X is nonempty, we either remove an element from X if all its successors are marked, or if there is an unmarked successor x of an element of X, then we mark x and add it to X. The algorithm finishes when the set X is empty. We prove that this algorithm marks

all nodes reachable from root using the relation next. Also we prove termination for this initial algorithm.

The second algorithm uses a stack of nodes instead of a set. In the first version of the algorithm, any element of the set could be used to proceed with marking, the stack version always chooses the element at the top of the stack. If all successors of the top are marked, then the top is removed, otherwise an unmarked successor of the top of the stack is marked and pushed onto the stack. The stack stores the path from the current node to the root node. We derive the second algorithm with a data refinement from the first algorithm.

A second data refinement step replaces the relation next by the collection $lnk.i$ of pointer functions and a function lbl which associate to every node a label from I. No extra variables are used for the stack, the stack is represented using the lnk and lbl variables. The data refinement shows that the new algorithm does the same computation as the previous algorithm, and that the variables lnk and lbl are restored to their initial values upon completion.

Finally, a third data refinement step replaces the general $lnk.i$ pointers with left and right pointers, and thus yields the classical DSW algorithm. The function lbl is also replaced by the atom bit.

Here we present some steps of the construction of the algorithm, but we do not give any proofs. All proof obligations for this algorithm have, however, been mechanically verified with the theorem prover Isabelle/HOL.

4.1 Marking using a set

In the first algorithm we start with a set X containing the root element. As long as the set X is non-empty we repeat the following steps: if there exists a unmarked successor node x of a node in X, then we mark x and we add it to X; or if there is a node x in X such that all successors of x are marked, then we remove x from X. The algorithm terminates when the set X is empty.

Initially we know that the marking bit of all nodes is false, and we also assume that there is a finite number of nodes. The assumption that we have a finite number of nodes will be used to prove the termination of the algorithm. In the final state the marking bit of a node is true if and only if this node is reachable from root. While marking, we maintain the invariant that all nodes in the set X are marked, all nodes marked so far are reachable, and for every reachable node x either x is marked or there exists a path of un-marked nodes from a node in X to x. The termination is given by the fact that at each step, we either mark a node and add it to the set X, or we remove a node from X. Therefore, at each step, the term $2 \cdot |\overline{mrk}| + |X|$ is decreased, where mrk is the set of currently marked nodes and $|\overline{mrk}|$ stands for the number of unmarked nodes.

The algorithm works on the following program variables and constants (that

together make up the abstract data structure):

$$\mathsf{nil} : \mathsf{node}$$

$$\mathsf{root} : \mathsf{node}$$

$$\mathsf{next} \subseteq \mathsf{node} \times \mathsf{node}$$

$$mrk \subseteq \mathsf{node}$$

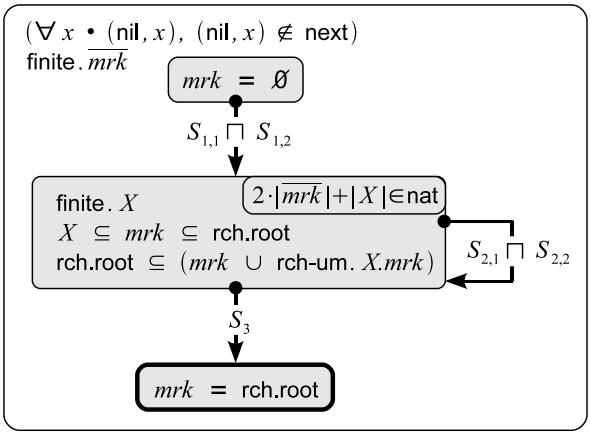
$$X \subseteq \mathsf{node}$$

The type node is the type of all graph nodes. In practice this type is the type of all memory addresses (pointers). The constant nil is the null pointer, root is the initial graph node from which we compute the reachable nodes, and next is the relation which gives the graph structure. The set mrk is the variable in which we compute the set of reachable nodes. Initially mrk is set to empty-set, and on completion it will hold the set of all nodes reachable from root. The set X is an auxiliary variable which is initialized to singleton root. We define the following auxiliary functions:

$$\begin{aligned} \mathsf{rch.root} &= \{x \mid (\mathsf{root}, x) \in \mathsf{next}^*\} \\ \mathsf{rch\text{-}um}.X.mrk &= \{x \mid (\exists y \in X \bullet (y, x) \in \mathsf{next} \circ (\mathsf{next} \cap \overline{mrk} \times \overline{mrk})^*)\} \end{aligned}$$

where $\overline{mrk}$ is the set complement of mrk ($\mathsf{node} - mrk$), and for a relation R, R^* is the reflexive and transitive closure of R. The set rch.root contains all nodes reachable from root and $\mathsf{rch\text{-}um}.X.mrk$ contains all nodes reachable from X along unmarked nodes. The predicate $\mathsf{finite}.X$ is true if the set X is finite.

The following invariant diagram solves the marking problem in terms of the data representation above.



(9)

We define the statements in the diagram as follows:

$$
\begin{aligned}
S_{1,1} &= [\,\mathsf{root} = \mathsf{nil}\,]\,;\, X := \emptyset \\
S_{1,2} &= [\,\mathsf{root} \neq \mathsf{nil}\,]\,;\, X := \{\mathsf{root}\}\,;\, mrk := \{\mathsf{root}\}] \\
Q_{2,1} &= (\lambda X, mrk, X', mrk' \bullet (\exists x \in X,\ \exists y \notin mrk \bullet \\
&\qquad (x,y) \in \mathsf{next} \wedge X' = X \cup \{y\} \wedge mrk' = mrk \cup \{y\})) \\
Q_{2,2} &= (\lambda X, mrk, X', mrk' \bullet (\exists x \in X \bullet \\
&\qquad (\forall y \bullet (x,y) \in \mathsf{next} \Rightarrow y \in mrk) \wedge X' = X - \{x\} \wedge mrk' = mrk)) \\
S_{2,1} &= [X,\ mrk := X',\ mrk' \,|\, Q_{2,1}.X.mrk.X'.mrk'] \\
S_{2,2} &= [X,\ mrk := X',\ mrk' \,|\, Q_{2,2}.X.mrk.X'.mrk'] \\
S_3 &= [\,X = \emptyset\,]
\end{aligned}
$$

Total correctness of the above diagram is reduced to the following proof obligations:

$$\mathsf{init}.mrk \;\{\!|\, S_{1,k} \,|\!\}\; \mathsf{inv}.X.mrk$$

$$\mathsf{inv}.X.mrk.w \;\{\!|\, S_{2,k} \,|\!\}\; \mathsf{inv}.X.mrk.(< w)$$

$$\mathsf{inv}.X.mrk \;\{\!|\, S_3 \,|\!\}\; \mathsf{final}.mrk$$

for all $k \in \{1, 2\}$, where

$$
\begin{aligned}
\mathsf{init}.mrk &= (\forall x \bullet (\mathsf{nil}, x),\ (x, \mathsf{nil}) \notin \mathsf{next}) \wedge \mathsf{finite}.\overline{mrk} \wedge mrk = \emptyset \\
\mathsf{inv}.X.mrk &= (\forall x \bullet (\mathsf{nil}, x),\ (x, \mathsf{nil}) \notin \mathsf{next}) \wedge \mathsf{finite}.\overline{mrk} \wedge \mathsf{finite}.X \\
&\qquad \wedge X \subseteq mrk \subseteq \mathsf{rch}.\mathsf{root} \wedge \mathsf{rch}.\mathsf{root} \subseteq (mrk \cup \mathsf{rch\text{-}um}.X.mrk) \\
\mathsf{inv}.X.mrk.w &= \mathsf{inv}.X.mrk \wedge 2 \cdot |\overline{mrk}| + |X| = w \\
\mathsf{final}.mrk &= (\forall x \bullet (\mathsf{nil}, x),\ (x, \mathsf{nil}) \notin \mathsf{next}) \wedge \mathsf{finite}.\overline{mrk} \wedge mrk = \mathsf{rch}.\mathsf{root}
\end{aligned}
$$

We proved all these proof obligations in Isabelle/HOL theorem prover

4.2 Marking using a stack

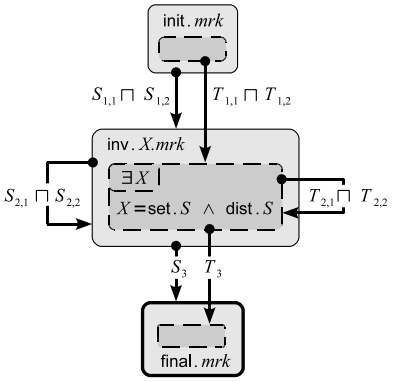
Instead of X, we now start with a stack (list) S containing the root element. As long as the stack is non-empty we repeat the following step: If h is the top element of S and there exists an unmarked node y such that $(h, y) \in \mathsf{next}$, then we mark y and we add it to the top of the stack. Otherwise we pop the top element of the stack. The algorithm terminates when the stack is empty.

If S is a list with elements from node, then $\mathsf{hd}.S$ it the first (top) element of the list and $\mathsf{tl}.S$ is the list obtained from S by removing the first element. We call $\mathsf{hd}.S$

and $\mathsf{tl}.S$ the *head* and the *tail* of S, respectively. If S is empty, then $\mathsf{hd}.S$ is nil and $\mathsf{tl}.S$ is the empty list. We denote by $[\,]$ the empty list, $[x]$ the list with one element, and we use $+$ for list concatenation. The predicate $\mathsf{dist}.S$ is true if all elements of S are distinct of each other and the function set applied to S gives the set containing the elements of S.

The set variable X from the abstract program is replaced by a variable S which is a list of distinct elements. The data abstraction invariant states that the set X is equal to $\mathsf{set}.S$. All other variables from the abstract program are present in the concrete program with the same meaning.

The diagram for this refinement is:



(10)

where

$$
\begin{aligned}
T_{1,1} &= [\,\mathsf{root} = \mathsf{nil}\,]\,;\ S := [\,]\\
T_{1,2} &= [\,\mathsf{root} \neq \mathsf{nil}\,]\,;\ S := [\mathsf{root}]\ ;\ mrk := \{\mathsf{root}\}\\
T_{2,1} &= [S,\ mrk := S',\ mrk' \mid S \neq [\,] \wedge (\exists y \notin mrk \bullet (\mathsf{hd}.S, y) \in \mathsf{next}\\
&\qquad\qquad \wedge\ mrk' = mrk \cup \{y\} \wedge S' = [\,y\,] + S)]\\
T_{2,2} &= [\,S \neq [\,] \wedge (\forall y \bullet (\mathsf{hd}.S, y) \in \mathsf{next} \Rightarrow y \in mrk)]\ ;\ S := \mathsf{tl}.S\\
T_3 &= [\,S = [\,]\,]
\end{aligned}
$$

The proof obligations for this data refinement step, after some simplifications, are

$$\mathsf{init}.mrk \wedge mrk = mrk_0 \,\{\!|\, T_{1,1} \,|\!\}\, \mathsf{root} = \mathsf{nil} \wedge \mathsf{dist}.S \wedge \mathsf{set}.S = \emptyset$$

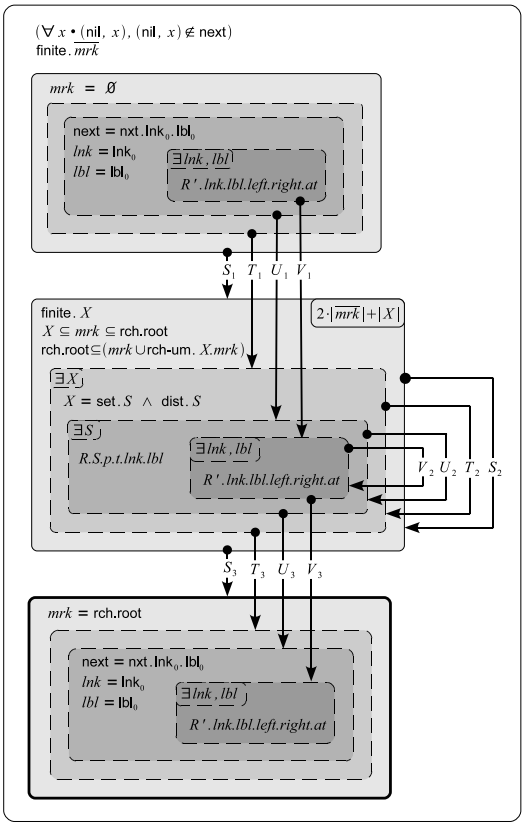
$$\mathsf{init}.mrk \wedge mrk = mrk_0 \,\{\!|\, T_{1,2} \,|\!\}\, \mathsf{root} \neq \mathsf{nil} \wedge \mathsf{dist}.S \wedge \mathsf{set}.S = \{\mathsf{root}\} \wedge mrk = \{\mathsf{root}\}$$

$$\mathsf{dist}.S \wedge X_0 = \mathsf{set}.S \wedge \mathsf{inv}.X_0.mrk \wedge mrk = mrk_0 \,\{\!|\, T_{2,k} \,|\!\}\, \mathsf{dist}.S \wedge Q_{2,k}.X_0.mrk_0.(\mathsf{set}.S).mrk$$

$$\mathsf{dist}.S \wedge X_0 = \mathsf{set}.S \wedge \mathsf{inv}.X_0.mrk \wedge mrk = mrk_0 \,\{\!|\, T_3 \,|\!\}\, X_0 = \emptyset$$

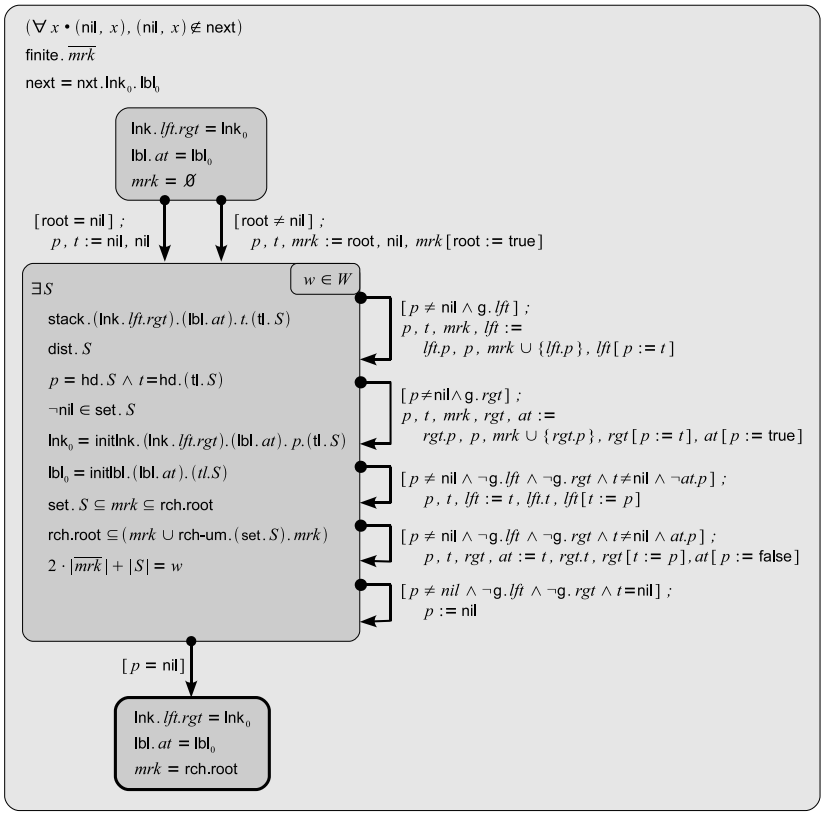
4.3 Complete derivation

The final marking algorithm is constructed using two additional data refinement steps. The following diagram collects all these refinement steps into a single diagram. The diagram shows explicitly all situations, and how they are built up as successive layers of abstraction. The transitions describe all successive versions of the marking algorithm. The diagram contains all the information that we need in order to verify that each version of the algorithm is correct (provided we also have access to the definitions used in expressing situations and transitions).



4.4 Final algorithm

Expanding definitions in the previous diagram and simplifying the constraints give us the following final invariant diagram for the DSW marking algorithm.



The main variables used in this final concrete program are *lft*, *rgt*, *at*, and *mrk*. The variables *lft* and *rgt* store the left and right successors of a graph node. The variable *at* defines what nodes are marked and is used during marking, and *mrk* contains the set of all reachable nodes at the end. The variables *lft*, *rgt*, and *at* are altered during marking, but they are restored to their initial values when the program terminates. The program uses also two auxiliary variables p, t : node. Variable p stores the head of the stack S, and t stores the head of the tail of S. The rest of S is stored by temporarily modifying the variables *lft*, *rgt* and *at*.

The invariants use a number of predicates and functions that we will not discuss further here. Their formal definitions are given below.

$$lft,\ rgt : \mathsf{node} \to \mathsf{node}$$

$$p,\ t : \mathsf{node}$$

$$at : \mathsf{node} \to \mathsf{bool}$$

$$A = \{\mathsf{none}, \mathsf{some}\} \text{ where } \mathsf{none} \neq \mathsf{some} \text{ are two labels}$$

$$\mathsf{lnk}_0 : A \to \mathsf{node} \to \mathsf{node}$$

$$\mathsf{lbl}_0 : \mathsf{node} \to A$$

$$\mathsf{lnk}.lft.rgt.i = \begin{cases} lft & \text{if } i = \mathsf{none} \\ rgt & \text{if } i = \mathsf{some} \end{cases}$$

$$\mathsf{lbl}.at.x = \begin{cases} \mathsf{some} & \text{if } at.x \\ \mathsf{none} & \text{if } \neg at.x \end{cases}$$

$$\mathsf{nxt}.lnk.lbl = \{(x,y) \mid (\exists i \bullet lnk.i.x = y) \wedge x = \mathsf{nil} \wedge y \neq \mathsf{nil} \wedge lbl.x = \mathsf{none}\}$$

$$\mathsf{g}.ptr = ptr.p \neq \mathsf{nil} \wedge ptr.p \notin mrk \wedge \neg at.p$$

$$\begin{aligned}
\mathsf{stack}.lnk.lbl.x.[\,] &= (x = \mathsf{nil}) \\
\mathsf{stack}.lnk.lbl.x.([y] + S) &= x \neq \mathsf{nil} \wedge x = y \wedge x \notin \mathsf{set}.S \\
&\qquad \wedge \mathsf{stack}.lnk.lbl.(lnk.(lbl.x).x).S \\
\mathsf{initlnk}.lnk.lbl.p.[\,] &= lnk \\
\mathsf{initlnk}.lnk.lbl.p.([x] + S) &= \mathsf{initlnk}.(lnk[(lbl.x) := ((lnk.(lbl.x))[x := p])]).lbl.y.S \\
\mathsf{initlbl}.lbl.[\,] &= lbl \\
\mathsf{initlbl}.lbl.([x] + S) &= \mathsf{initlbl}.(lbl[x := \mathsf{none}]).S
\end{aligned}$$

5 Conclusions

We have in this paper shown how to carry out data refinement of invariant based programs, and have applied the technique to the construction of the classical Deutsch-Schorr-Waite graph marking algorithm. We have used predicate transformer semantics for the transitions in the invariant diagrams, and we have shown that termination is preserved by data refinement.

We have shown how the overall proof effort of the marking algorithm, is simplified by first proving a generalized version of it, and then data refining it to the classical version. All results presented in this paper have been proved mechanically using the Isabelle/HOL interactive theorem prover. This gives a very solid foundation of our results. Currently we are working on including data refinement into the Socos environment [6,7], which is specifically designed to support the construction of invariant based programs and proving their correctness.

References

[1] J.-R. Abrial. Event based sequential program development: Application to constructing a pointer program. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 51–74. Springer, 2003.

[2] R. J. Back. *Correctness preserving program refinements: proof theory and applications*, volume 131 of *Mathematical Centre Tracts*. Mathematisch Centrum, Amsterdam, 1980.

[3] R. J. Back. Semantic correctness of invariant based programs. In *International Workshop on Program Construction*, Chateau de Bonas, France, 1980.

[4] R. J. Back. Invariant based programs and their correctness. In W. Biermann, G Guiho, and Y Kodratoff, editors, *Automatic Program Construction Techniques*, pages 223–242. MacMillan Publishing Company, 1983.

[5] R. J. Back. Invariant based programming: Basic approach and teaching experience. *Formal Aspects of Computing*, 2008.

[6] R. J. Back, J. Eriksson, and M. Myreen. Verifying invariant based programs in the SOCOS environment. In P. Boca, J. P. Bowen, and D. A. Duce, editors, *Teaching Formal Methods: Practice and Experience*, Electronic Workshops in Computing (eWiC). BCS, Dec 2006.

[7] R. J. Back, J. Eriksson, and M. Myreen. Testing and verifying invariant based programs in the SOCOS environment. In *The International Conference on Tests And Proofs (TAP)*, 2007.

[8] R. J. Back and V. Preoteasa. Semantics and proof rules of invariant based programs. Technical Report 903, TUCS, Jul 2008.

[9] R. J. Back and J. von Wright. *Refinement Calculus. A systematic Introduction*. Springer, 1998.

[10] R. J. Back and J. von Wright. Encoding, decoding and data refinement. *Formal Aspects of Computing*, 12:313–349, 2000.

[11] W. DeRoever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, New York, NY, USA, 1999.

[12] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4), December 1972.

[13] D. E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[14] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.

[15] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[16] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.