

Global invariants for analysing multi-threaded applications

Helmut Seidl^a, Varmo Vene^b, and Markus Müller-Olm^c

^a Fachbereich IV– Informatik, Universität Trier, D-54286 Trier, Germany;
seidl@psi.uni-trier.de

^b Institute of Computer Science, University of Tartu, J. Liivi 2, 50409 Tartu, Estonia;
varmo@cs.ut.ee

^c Fachbereich Informatik, Lehrgebiet Praktische Informatik 5, FernUniversität Hagen, D-58084 Hagen, Germany; mmo@ls5.cs.uni-dortmund.de
On leave from Universität Dortmund

Received 28 January 2003, in revised form 16 July 2003

Abstract. We exhibit an interprocedural framework for the analysis of multi-threaded programs based on *partial invariants* of a new kind of constraint systems which we call *side-effecting*. We explore the formal properties of these constraint systems and provide general techniques for computing partial invariants. The practicality of this approach is demonstrated by designing and implementing a reasonably efficient flow- and context-sensitive interprocedural data-race analyser of multi-threaded C.

Key words: program analysis, multi-threading, constraint solving.

1. INTRODUCTION

The DAEDALUS project is a joint European technology transfer project with industrial and academic partners, which aims at applying techniques from abstract interpretation in order to improve avionics software. Our particular goal is to enhance reliability of multi-threaded C code by using program analyser technology for obtaining sanity checks or even certificates stating the absence of certain programming errors. This type of application does not demand analysers which run in a few seconds but flag thousands of unnecessary warnings which later on must be checked manually by highly paid software engineers. We are clearly willing to spend some minutes analysing larger programs, provided that the number of spurious errors is dramatically decreased. Therefore, we aim at a good balance between precision and analysis time.

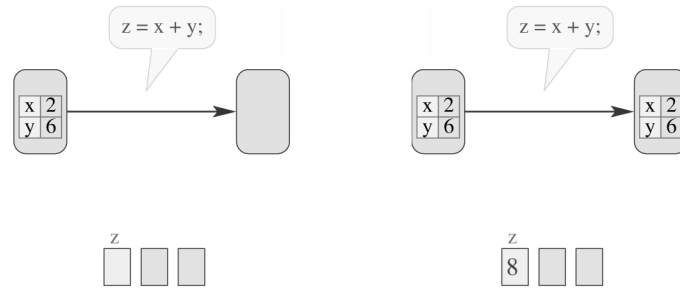


Fig. 1. An example control-flow edge and the effect of traversing the edge.

The analysis of multi-threaded programs has been considered as notoriously difficult and expensive. In fact, *precise* analyses are known for some restricted classes of parallel programs [1–3] but for very simple program properties only [4]. However, in order to arrive at the necessary precision for a nontrivial fragment of C, we have, e.g., to resolve function pointers and integrate some form of *points-to* analysis. Also, we have to take into account the possible interference between the execution of different threads. In this paper, we present the background concepts which we have used in our generator for interprocedural analyses of multi-threaded C in order to arrive at sufficiently precise and efficient analyses. The key observation is that POSIX threads communicate through global variables. In order to separate the analysis of the different threads, we attempt to infer for each global variable one value which safely approximates *all* possible states of the global variable. This single invariant for the globals is then used for analysing each thread individually. Consider, e.g., the control-flow edge shown on the left in Fig. 1. There, the global int variable z is updated to the sum of two local variables. The desired result of applying the edge’s transfer function is shown on the right in Fig. 1. Traversing the edge has *two* effects conceptually: first, the local state of the edge’s target node receives the local variable assignment from the edge’s source node, and second, the global z receives the value of the expression $x + y$. Given that some global invariant for z safely approximates the second effect onto the global, the analysis may proceed by tracking the local states only. In the presence of multi-threading, this means that we can analyse each thread individually. The separation is particularly successful in applications where the threads are only loosely coupled, i.e., where the control flow mainly depends on the values of locals. We are left with the task of inferring an as tight invariant as possible. The basic idea for this approximation is to refine a *partial invariant* during the fixpoint iteration by collecting the *side-effects* of constraint evaluation.

In earlier work, we have successfully applied general *local* constraint solvers to implementing interprocedural analyses [5–9]. The architectural idea is to separate the fixpoint engine completely from the program analysis framework such that each of these two software components can be developed, optimized, and exchanged

independently of the other. We may, e.g., experiment with interprocedural analyses based on the functional approach or using call-strings of various lengths [10] without changing the underlying fixpoint engine. This separation of concerns and subdivision into small software components is even more desirable for a software checking application, where not a single programming error can be tolerated.

In this paper, we generalize this approach to an analysis framework for multi-threaded programs and partial invariants. This framework allows us to specify and implement conveniently, e.g., data-race analysers for multi-threaded C. It is based on partial invariants over a class of generalized constraint systems which we call *side-effecting*. We give an application-independent characterization of partial invariants by means of ordinary constraint systems and present a sufficient condition guaranteeing the existence of a unique least partial invariant. Inferring a least partial invariant means for the analysis that only those effects onto globals are recorded, which occurred during tracking function calls. Finally, we show how *local* constraint solving can be customized to infer partial invariants. We have implemented this approach and report about our experimental results.

2. THE MULTI-THREADED INTERPROCEDURAL FRAMEWORK

In this section, we present our framework for interprocedural analysis of multi-threaded programs. We do so by using the following C program as our running example:

```

int z;
mutex A, B;

void inc(mutex* me) {
    mutex_lock(me);
    z = z + 1;
    mutex_unlock(me);
}

void main() {
    tid id;

    z = 0;
    create(&id,inc,&A);
    inc(&A);
}

```

Here, `create()`, `mutex_lock()`, and `mutex_unlock()` are simplified versions of the corresponding pthread library functions. The first argument of `create()` is the address of the variable where the thread id of the newly created thread is placed, the second argument contains (a reference to) the function to be executed by the new thread, and the third argument contains the actual parameter to the called function. The mutex handling functions have been simplified to receive the address of a mutex only.

In general, we assume that every input program consists of a finite set \mathcal{F} of functions, and that each function $p \in \mathcal{F}$ is specified via a finite control-flow graph representing the body of the function. The control-flow graphs corresponding to our example program are depicted in Fig. 2.

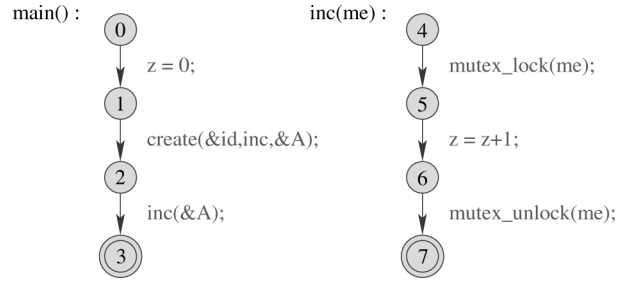


Fig. 2. The example control-flow graph.

2.1. Specifying analyses

Assume that the abstract local state of a thread is described by elements from a lattice \mathbb{D}_1 , whereas the global state, i.e., the part of the program execution state which is accessible to more than one thread, is described by elements from another lattice \mathbb{D}_2 . Typically, the latter is of the form $\mathbb{D}_2 = G \rightarrow \mathbb{D}$, where G is the set of global variables and abstract heap locations and \mathbb{D} is a lattice of abstract values for the globals.

Now, the analysis can be specified by assigning a transfer function to every edge in the control-flow graph which implements the (abstract) effect of traversing this edge during program execution. In the simplest case, when the edge $e = (u, v)$ performs a basic computation step (e.g., an assignment or calls of externally defined functions), the abstract effect is described by a function \mathbf{trans}_e :

$$\mathbf{trans}_e : \mathbb{D}_1 \times \mathbb{D}_2 \rightarrow \mathbb{D}_1 \times \mathbb{D}_2.$$

The function \mathbf{trans}_e takes the local and global state and returns the new local state. In principle, it additionally returns the complete new global state. However, in the case that $\mathbb{D}_2 = G \rightarrow \mathbb{D}$ we will later also consider a “differential” formulation, where we return only values of those globals which are potentially modified (the “side-effect”).

Next, assume that the edge $e = (u, v)$ represents a call of a locally defined function. Then, the abstract effect is determined relative to two functions, \mathbf{entry}_e and $\mathbf{combine}_e$:

$$\begin{aligned} \mathbf{entry}_e & : \mathbb{D}_1 \times \mathbb{D}_2 \rightarrow \mathbb{D}_1 \times 2^{\mathcal{F}} \\ \mathbf{combine}_e & : \mathbb{D}_1 \times \mathbb{D}_1 \rightarrow \mathbb{D}_1. \end{aligned}$$

The function \mathbf{entry}_e again takes the local state and global state as its arguments and returns the starting local state for the body of the called function. This suffices if the function to be called is statically known. In realistic C code, however, function calls may happen through pointers implying that we do not necessarily know the functions to be called in advance. Instead, they may depend on the local or global state. Due to potential loss in precision, it may even happen that the best we can

tell is that the function is contained in a set of *possibly* called functions. In order to take care of this, we let the function entry_e additionally return a set of functions which are potentially called at e . The function combine_e merges the effect of the called functions with the local state of the caller.

Finally, consider an edge $e = (u, v)$ where new threads are created. Then the abstract effect is determined relative to a function create_e :

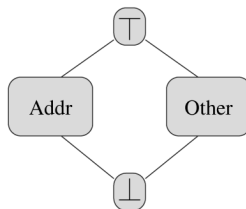
$$\text{create}_e : \mathbb{D}_1 \times \mathbb{D}_2 \rightarrow \mathbb{D}_1 \times \mathbb{D}_2 \times \mathbb{D}_1 \times 2^{\mathcal{F}}.$$

The function create_e takes the local state and global state and returns a new local state together with a new global state (e.g., storing the abstract id of the newly created thread). Additionally, it provides the initial local state for the created threads. Similar to the case of call edges, the function to be called by the newly created thread, may depend on the local or global state. Accordingly, the function create_e provides us with a (safe super-)set of functions potentially executed by threads created at this edge.

2.2. A simple data-race analysis

As an example, consider a simple data-race analysis for detection of potentially unsafe accesses to global resources. In multi-threaded C, accesses to global variables should be protected by mutexes. In the simplest case, each use of a global variable should be protected by the same mutex. Thus, the analysis has to determine which mutexes are *definitely* locked at a given point. However, at the beginning of the execution of the program, when no threads are created yet, it is safe to modify globals (e.g., for initialization purposes) without locking the corresponding mutex. Hence, in order to minimize the number of false alarms, the analysis should keep track of whether any thread has already been created or not. In addition, the analysis needs a base constant propagation, at least for disambiguating the address arguments. Therefore, the local state maintained during the analysis consists of three components: the first holds the set of definitely held mutex locks, the second contains a flag whether any thread has been created or not, and the third records the (abstract) values of the local variables.

The domain of abstract values should (at least) provide abstract descriptions of addresses of variables (for resolving function calls as well as variable assignments) together with abstract descriptions of potential values. For our example program, it suffices to consider a lattice of the following structure:



Addr and Other represent partial orderings for the description of addresses and other values, respectively. We choose Addr as the lattice of all subsets of some set Base of base addresses. In our minimalistic example, Base is given by

$$\text{Base} = \{\text{null}, \&A, \&B, \&\text{id}, \&x, \&z, \&\text{main}, \&\text{inc}\}.$$

We allow sets of addresses in order to avoid as many unknown pointer dereferences as possible. The ordering \sqsubseteq on the sublattice Addr is subset inclusion \subseteq .

For the globals, we record at each variable the set of definitely held mutexes when accessing the variable together with an abstract value. For collecting locked mutexes, we use the same lattice as for ordinary addresses – this time, however, with the ordering reversed (as we are interested in definite locks). In particular, the least element of this lattice is given by the universal set of mutexes and the greatest element equals \emptyset .

Having specified the involved domains, we can now define the abstract behaviour of our example program (see Fig. 3). Functions $\text{trans}_{(0,1)}$ and $\text{trans}_{(5,6)}$ correspond to assignments to the global z . As z is the only (either local or global) variable affected by these transitions, the local state is returned unchanged and the global state is updated for z . The new global state for z consists of the set of definitely held mutexes at the current program point together with the new abstract value (which “lives” in the Other sublattice of the domain of abstract values). However, if there had been no thread creations before (i.e., the second component of the local state is `false`), then we let the mutex set equal \perp as the access for z is guaranteed to be safe.

Functions $\text{trans}_{(4,5)}$ and $\text{trans}_{(6,7)}$ correspond to the calls of external functions `mutex_lock()` and `mutex_unlock()`, respectively. Both transfer functions modify the current mutex set M and leave the other components of the local state unchanged. As no global variable is affected by these transitions, the global state is left unchanged. In the case of `mutex_lock()`, the current abstract value of the variable `me` is added to the set M using the auxiliary function `add`. As we are interested in definitely held mutexes, the function `add` extends the set M only if the abstract value $\rho(\text{me})$ consists of a singleton address set. If $\rho(\text{me})$ consists of more than one address, M is left unchanged. In the case of `mutex_unlock()`, the abstract value $\rho(\text{me})$ is subtracted from the set M using the auxiliary function `rem`. In this case, all addresses contained in $\rho(\text{me})$ are removed from M .

Functions $\text{entry}_{(2,3)}$ and $\text{combine}_{(2,3)}$ correspond to the call of the function `inc`. The function $\text{entry}_{(2,3)}$ returns the initial local state of the call which binds the formal parameter `me` to the abstract value $\{\&A\}$. In addition, it returns a set of possibly called functions which contains `inc` as sole element. As the call to `inc` does not modify any locals, the function $\text{combine}_{(2,3)}$ ignores these “modifications” and returns the current local state.

Finally, the function $\text{create}_{(1,2)}$ creates a new local state, where the mutex set remains unchanged but the thread creation flag is set to `true` and local variable `id` receives the value \top , since we do not know which thread `id` is created by `create()`.

$$\begin{aligned}
\mathbf{trans}_{(0,1)}(d, \tau) &= \mathbf{let} \quad \langle M, f, _ \rangle = d \\
&\quad M' = \mathbf{if} \ f \ \mathbf{then} \ M \ \mathbf{else} \ \perp \\
&\quad \mathbf{in} \quad (d, \tau[z \mapsto \langle M', 0 \rangle]) \\
\mathbf{create}_{(1,2)}(d, \tau) &= \mathbf{let} \quad \langle M, f, \rho \rangle = d \\
&\quad d' = \langle M, \mathbf{true}, [\mathbf{id} \mapsto \top] \rangle \\
&\quad d'' = \langle \emptyset, \mathbf{true}, [\mathbf{me} \mapsto \{\&A\}] \rangle \\
&\quad \mathbf{in} \quad (d', \tau, d'', \{\mathbf{inc}\}) \\
\mathbf{entry}_{(2,3)}(d, \tau) &= \mathbf{let} \quad \langle M, f, _ \rangle = d \\
&\quad d' = \langle M, f, [\mathbf{me} \mapsto \{\&A\}] \rangle \\
&\quad \mathbf{in} \quad (d', \{\mathbf{inc}\}) \\
\mathbf{combine}_{(2,3)}(d, d') &= d \\
\mathbf{trans}_{(4,5)}(d, \tau) &= \mathbf{let} \quad \langle M, f, \rho \rangle = d \\
&\quad M' = \mathbf{add}(M, \rho(\mathbf{me})) \\
&\quad d' = \langle M', f, \rho \rangle \\
&\quad \mathbf{in} \quad (d', \tau) \\
\mathbf{trans}_{(5,6)}(d, \tau) &= \mathbf{let} \quad \langle M, f, _ \rangle = d \\
&\quad M' = \mathbf{if} \ f \ \mathbf{then} \ M \ \mathbf{else} \ \perp \\
&\quad \langle _, v \rangle = \tau(z) \\
&\quad \mathbf{in} \quad (d, \tau[z \mapsto \langle M', v + 1 \rangle]) \\
\mathbf{trans}_{(6,7)}(d, \tau) &= \mathbf{let} \quad \langle M, f, \rho \rangle = d \\
&\quad M' = \mathbf{rem}(M, \rho(\mathbf{me})) \\
&\quad d' = \langle M', f, \rho \rangle \\
&\quad \mathbf{in} \quad (d', \tau)
\end{aligned}$$

Fig. 3. Behavioural functions for the example program.

It also creates an initial local state for the newly created thread, where the mutex set is empty and the formal parameter `me` is bound to `{&A}`. The global state remains unchanged and the set of possible start functions executed by the created thread has `inc` as only element.

2.3. The overall approach

The specification of the analysis still allows us to model the exact (abstract) multi-threaded program execution with interleaving semantics – given that globals are modified atomically at control-flow edges. For large programs or high degrees of parallelism this, however, is not feasible. What is needed here is a technique to decouple the analysis of the involved threads in order to avoid state explosion. Therefore, we aim at replacing the inspection of the Cartesian product of state

spaces with their sum. The basic idea to achieve this is to approximate all possibly occurring global states by one safe *global invariant*. Given this invariant, it suffices for the analysis to maintain information only about the local states of program points and procedure calls.

Accordingly, the set V of variables maintained by the analysis consists of:

Nodes: $\langle v, d \rangle$, v a program point of a function, $d \in \mathbb{D}_1$ the abstract entry value of the currently analysed instance of the function;

Calls: $\langle p, d \rangle$, p a function, $d \in \mathbb{D}_1$ an abstract entry value.

The specification of an analysis induces a set of constraints on the values of these variables. Given a correct invariant, these constraints should specialize to the classical constraints of the functional approach for interprocedural analysis [9,10]. Thus, we choose constraints of the form $x \leftarrow f$, where $x \in V$ and f is of type

$$f : (V \rightarrow \mathbb{D}_1) \times \mathbb{D}_2 \rightarrow \mathbb{D}_1 \times \mathbb{D}_2 \times 2^V.$$

We call constraint systems of this form *side-effecting*. In our application, the right-hand side f takes an assignment $\sigma : V \rightarrow \mathbb{D}_1$ for program points and procedure calls together with a global state $\tau : \mathbb{D}_2$ and returns a triple (b, η, s) consisting of a local state b for the left-hand side together with an updated global state η and a set s of calls which are executed by possibly spawned threads.

The constraints are determined relative to the behavioural functions for the edges of the control-flow graph. Every edge $e = (u, v)$ induces one constraint for every local state $d \in \mathbb{D}_1$, as follows:

- If e is a basic edge,
 $\langle v, d \rangle \leftarrow \lambda(\sigma, \tau). \mathbf{let} (b, \eta) = \mathbf{trans}_{(u,v)}(\sigma\langle u, d \rangle, \tau)$
 $\mathbf{in} (b, \eta, \emptyset)$
- If e is a call edge,
 $\langle v, d \rangle \leftarrow \lambda(\sigma, \tau). \mathbf{let} (b_1, F) = \mathbf{entry}_{(u,v)}(\sigma\langle u, d \rangle, \tau)$
 $b = \bigsqcup \{ \mathbf{combine}_{(u,v)}(\sigma\langle u, d \rangle, \sigma\langle p, b_1 \rangle) \mid p \in F \}$
 $\mathbf{in} (b, \lambda z. \perp, \emptyset)$
- If e is a create edge,
 $\langle v, d \rangle \leftarrow \lambda(\sigma, \tau). \mathbf{let} (b, \eta, b_1, F) = \mathbf{create}_{(u,v)}(\sigma\langle u, d \rangle, \tau)$
 $s = \{ \langle p, b_1 \rangle \mid p \in F \}$
 $\mathbf{in} (b, \eta, s)$

We also have a constraint to make the local state of a call available to the computation inside the instance of the called function. This means for every entry point u of some function, we have:

$$\langle u, d \rangle \leftarrow \lambda(\sigma, \tau). \langle d, \tau, \emptyset \rangle.$$

Finally, we have the following constraint for each call variable $\langle p, d \rangle$, where r is the return point of function p :

$$\langle p, d \rangle \leftarrow \lambda(\sigma, \tau). \langle \sigma \langle r, d \rangle, \tau, \emptyset \rangle.$$

In the next section, we make precise what we mean by the notions “(partial) invariant” and “solution” of such constraint systems.

3. SIDE-EFFECTING CONSTRAINT SYSTEMS

For the following, let V denote a set of constraint variables. Let $x \leftarrow f$ denote a constraint, where the right-hand side is a function of type

$$f : (V \rightarrow \mathbb{D}_1) \times \mathbb{D}_2 \rightarrow \mathbb{D}_1 \times \mathbb{D}_2 \times 2^V.$$

We say that the assignment $\sigma : V \rightarrow \mathbb{D}_1$ *satisfies* the constraint $x \leftarrow f$ relative to a global state $\tau \in \mathbb{D}_2$ iff for $f(\sigma, \tau) = (d, \eta, s)$, $d \sqsubseteq \sigma x$, and also $\eta \sqsubseteq \tau$. Accordingly, a *complete* solution of a set \mathcal{C} of constraints relative to a global state τ is a mapping $\sigma : V \rightarrow \mathbb{D}_1$ satisfying *all* constraints in \mathcal{C} relative to τ . A global state τ is called an *invariant* of \mathcal{C} if there exists a complete solution of \mathcal{C} relative to τ . Using invariants and complete solutions is adequate if we use call-strings of finite length to distinguish function calls. In this case, approximations to possibly occurring actual parameters are propagated from the call sites to the entry points of functions.

If we use, however, the functional approach to interprocedural analysis (i.e., function bodies are analysed for every possible argument independently), such invariants and *complete* solutions are no longer sufficient. The only invariant of our example system from Section 2, e.g., maps the variable z to $\langle \emptyset, \top \rangle$, since this value must be safe for *all* calls of the function `inc`. Note that the variables from V represent the set of all *formally possible* local program configurations – even those which are in fact unreachable (like calling `inc` with the mutex `B`). The invariant, though, is only needed for configurations which are reachable. The smaller the set of program configurations we must consider, the more precise we can choose the invariant of the system, i.e., the more precise a result we return.

Reachability has been considered, e.g., in [6,8] for “ordinary” constraint systems, i.e., those where right-hand sides return values. There it was observed that the set of reachable program configurations closely corresponds to the set of fixpoint variables of the “natural” constraint system of the program when locally explored through a *demand-driven* fixpoint algorithm which returns a *partial* solution only. Here, we generalize this approach to side-effecting constraint systems with global invariants.

Let $\underline{\mathbb{D}}_1$ denote the complete lattice, which we obtain from \mathbb{D}_1 by adding \perp as a new least element (“superbottom”). The *partial variable assignments* from V to

\mathbb{D}_1 are given as the set of assignments $V \rightarrow \mathbb{D}_1$. The *domain*, $\text{dom } \sigma$, of a partial assignment is given by the set of all variables $x \in V$ with $\sigma x \neq \perp$.

We extend the function f to operate also on partial variable assignments. In order to do so, we need to determine the set variables *accessed* during the evaluation of f . In general, this set itself may depend on the values of variables. Therefore, it is described by a function

$$D_f : (V \rightarrow \mathbb{D}_1) \times \mathbb{D}_2 \rightarrow 2^V.$$

This function describes a property of the operational behaviour when evaluating f and therefore cannot (easily) be extracted from the denotational semantics of f . It is straightforward, however, to extract D_f if f is specified in some kind of expression language as in our framework or, even, to determine its values “at runtime”, i.e., by instrumenting the evaluation of f – this is what we will do when constructing a local constraint solver. The function D_f has the following property: if $D_f(\sigma, \tau) = X \subseteq V$, then f returns the same value on every pair (σ_1, τ) , where $\sigma_1 x = \sigma x$ for $x \in X$ (and $\sigma_1 x$ arbitrary otherwise).

When applying f to a partial variable assignment which is “underspecified”, i.e., does not provide non- \perp -values for as many variables as necessary to evaluate f , we return \perp . Thus, $f(\sigma, \tau) = (\perp, \perp, \emptyset)$, whenever $\sigma(x) = \perp$ for some $x \in D_f(\sigma, \tau)$.

Let (τ, X) denote a pair consisting of a global state $\tau \in \mathbb{D}_2$ and a subset $X \subseteq V$. A partial variable assignment $\sigma : V \rightarrow \mathbb{D}_1$ is called a *partial solution* of the constraint system \mathcal{C} relative to (τ, X) iff X is the domain of σ and for every constraint $x \leftarrow f$ with $(d, \eta, s) = f(\sigma, \tau)$ the following holds:

- $d \sqsubseteq \sigma x$, $\eta \sqsubseteq \tau$, and also
- $s \cup D_f(\sigma, \tau) \subseteq X$.

In particular, σ must be defined for all variables from s .

The pair (τ, X) is called a *partial invariant* of \mathcal{C} if there exists a partial solution of \mathcal{C} relative to (τ, X) . Every constraint system with $V \neq \emptyset$ has at least two partial invariants. The first one holds for an *empty set* of reachable variables only and is given by (\perp, \emptyset) . The second extreme considers *all* fixpoint variables as reachable and provides no information about the global state, i.e., is given by (\top, V) . Both invariants are not very interesting. In our applications, we are given a subset $I \subseteq V$ of initial configurations which are trivially reachable and therefore must be included into the set of reachable configurations. In our framework, the set I is given by all possible initial calls of the program `main`. Thus, the goal is to determine an as small partial invariant (τ, X) as possible such that $I \subseteq X$. Note that, the smaller the invariant, the more precise an information is provided for the globals.

In our example, the set I is given by $\{\langle \text{main}, \top \rangle\}$, and there is a partial invariant (τ, X) such that τ maps the global z to $\langle \{\&A\}, \top \rangle$. Note that the second component of this value still is not very informative – quite in contrast to the extra property recorded in the first component, which assures that accesses to z are always protected by the mutex A .

In order to reason about the possible existence and uniqueness of partial invariants, we define for a given side-effecting constraint system \mathcal{C} and a subset $I \subseteq V$ of initial variables the corresponding *combined constraint* $E_{\mathcal{C},I}$:

$$(\sigma, \tau) \sqsupseteq F(\sigma, \tau) \sqcup (\sigma_I, \perp).$$

Here, $\sigma_I x = \perp$ if $x \in I$ and \perp otherwise, and the function

$$F : (V \rightarrow \mathbb{D}_1) \times \mathbb{D}_2 \rightarrow (V \rightarrow \mathbb{D}_1) \times \mathbb{D}_2$$

is given by $F(\sigma, \tau) = (\sigma_1, \tau_1)$, where

$$\begin{aligned} \sigma_1 x &= \bigsqcup \{d \mid \sigma x \neq \perp \wedge \exists x \leftarrow f \in \mathcal{C} : (d, _, _) = f(\sigma, \tau)\} \\ &\sqcup \bigsqcup \{\perp \mid \exists y \leftarrow f \in \mathcal{C} : \sigma y \neq \perp \\ &\quad \wedge (_, _, s) = f(\sigma, \tau) \wedge (x \in s \cup D_f(\sigma, \tau))\}, \\ \tau_1 &= \bigsqcup \{\eta \mid \exists y \leftarrow f \in \mathcal{C} : \sigma y \neq \perp \wedge (_, \eta, _) = f(\sigma, \tau)\}. \end{aligned}$$

Only those constraints have a nontrivial contribution onto the output (σ_1, τ_1) whose left-hand sides have already a value $\neq \perp$. Accordingly, whenever a constraint f contributes to the result, all variables accessed during this evaluation receive a non- \perp -value. These two features will allow us to design a local solver which explores fixpoint variables backward through variable dependences.

The right-hand side of the constraint $E_{\mathcal{C},I}$ returns non- \perp -values for all $x \in I$. Hence, the (trivial) least solution $(\lambda x. \perp, \perp)$ is ruled out whenever $I \neq \emptyset$.

Proposition 1. *For a global state $\tau \in \mathbb{D}_2$ and a partial variable assignment σ with a domain X subsuming I , the following are equivalent:*

1. σ is a partial solution of \mathcal{C} relative to (τ, X) ;
2. (σ, τ) is a partial solution of $E_{\mathcal{C},I}$.

For later use, let us observe that the set of partial invariants as well as the set of relative partial solutions of a side-effecting constraint system \mathcal{C} remains the same if we replace the treatment of the global state by a “differential” one. In particular, when a constraint does not affect the global state, we can simply return \perp in the second component. More generally, assume we are given another constraint system \mathcal{C}' whose constraints are in one-to-one correspondence with those of \mathcal{C} such that the constraint $x' \leftarrow f' \in \mathcal{C}'$ corresponding to $x \leftarrow f \in \mathcal{C}$ has the following properties: $x' = x$, and

$$\begin{aligned} f(\sigma, \tau) &= \mathbf{let} (d, \eta, s) = f'(\sigma, \tau) \\ &\quad \mathbf{in} (d, \eta \sqcup \tau, s). \end{aligned}$$

Then the following holds:

Proposition 2. *For any partial variable assignment $\sigma : V \rightarrow \mathbb{D}_1$ with domain $X \supseteq I$ and global state $\tau \in \mathbb{D}_2$, the pair (σ, τ) is a partial solution of $E_{\mathcal{C},I}$ iff it is a partial solution of $E_{\mathcal{C}',I}$.*

While Proposition 2 will help us to construct efficient algorithms, Proposition 1 is the justification for applying fixpoint methods and provides sufficient conditions for the existence of unique least partial invariants.

Obviously, if F is monotonic, then it has a least solution by the Knaster–Tarski fixpoint theorem. In particular, this is the case when all functions f and D_f are monotonic (with respect to the obvious orderings). As observed in [6,8], however, interprocedural analyses usually do *not* introduce monotonic constraints. The abstract effect of a function p conceptually can be thought of as described by a (monotonic) function. Only in very simple cases, though, this function can be treated as a whole. Following, e.g., [10] we have replaced this function in our framework by the *set* of variables $\langle p, a \rangle$, $a \in \mathbb{D}_1$, each of which describes the results of the *abstract function call* of p on the abstract value a . The hope here is that only few of these (potentially many) calls are actually queried during fixpoint iteration. The constraint system for these variables is no longer monotonic – even if all behavioural functions trans_e , entry_e , combine_e , and create_e are. Instead, it is just *weakly* monotonic.

Weak monotonicity is defined relative to a partial ordering on the set of variables V . In case of the variable set used by our framework, this ordering is given by

$$\langle r_1, a_1 \rangle \leq \langle r_2, a_2 \rangle \text{ iff } r_1 = r_2 \text{ and } a_1 \sqsubseteq a_2.$$

A system \mathcal{C} of constraints (over V , \mathbb{D}_1 , and \mathbb{D}_2) is called *weakly monotonic* with respect to the partial ordering “ \leq ” iff the following properties hold:

1. For every constraint $x \leftarrow f$ in \mathcal{C} and every two variable assignments σ_1, σ_2 , where at least one of the σ_i is monotonic, and global states $\tau_i \in \mathbb{D}_2$, then $\sigma_1 \sqsubseteq \sigma_2$ and $\tau_1 \sqsubseteq \tau_2$ implies

$$f(\sigma_1, \tau_1) \sqsubseteq f(\sigma_2, \tau_2) \quad \text{and} \quad D_f(\sigma_1, \tau_1) \sqsubseteq D_f(\sigma_2, \tau_2).$$

2. For every constraint $x_1 \leftarrow f_1$ in \mathcal{C} and $x_2 \in X$ with $x_1 \leq x_2$, there is some constraint $x_2 \leftarrow f_2$ in \mathcal{C} such that $f_1 \sqsubseteq f_2$, i.e., for every pair of variable assignments (σ, τ) ,

$$f_1(\sigma, \tau) \sqsubseteq f_2(\sigma, \tau) \quad \text{and} \quad D_{f_1}(\sigma, \tau) \sqsubseteq D_{f_2}(\sigma, \tau),$$

whenever σ is monotonic.

These two properties enforce the natural conditions on monotonic assignments. Moreover, they allow us to relate the results on nonmonotonic assignments to comparable monotonic ones. Some remarks are appropriate here.

- A partial variable assignment σ , i.e., a mapping $\sigma : V \rightarrow \underline{\mathbb{D}}_1$, is considered as monotonic iff for all $x_1 \leq x_2$, $\sigma x_2 \neq \perp$ implies $\sigma x_1 \leq \sigma x_2$ (i.e., we demand nothing if $\sigma x_2 = \perp$).

- The pre-ordering on subsets $X_1, X_2 \subseteq V$ of variables is given by $X_1 \sqsubseteq X_2$ iff for all $x_1 \in X_1$, $x_1 \leq x_2$ for some $x_2 \in X_2$.

This is the pre-ordering induced by the subset ordering on the corresponding downward closed subsets of variables. In particular:

- We view two monotonic assignments $\sigma_1, \sigma_2 : V \rightarrow \mathbb{D}_1$ as *equivalent* if
 1. for any $x_1 \in V$, there is some $x_2 \in V$ with $x_1 \leq x_2$ such that $\sigma_1 x_1 \sqsubseteq \sigma_2 x_2$, and also
 2. for any $x_1 \in V$, there is some $x_2 \in V$ with $x_1 \leq x_2$ such that $\sigma_2 x_1 \sqsubseteq \sigma_1 x_2$.

If V has finite height, then monotonic assignments are equivalent iff they *agree* on variables which are maximal with respect to “ \leq ” in the union of their domains.

For a subset c of variables, let $c \downarrow$ denote its *downward closure*, i.e., the set of variables x' such that $x' \leq x$ for some $x \in c$. Let \mathbb{D}' denote the set of monotonic variable assignments in $V \rightarrow \mathbb{D}_1$ whose domain is downward closed. For later use, we make the following observation which essentially allows us to restrict attention to assignments from \mathbb{D}' and downward closed sets:

Proposition 3. *Assume \mathcal{C} is weakly monotonic, and σ is a partial solution of \mathcal{C} relative to (τ, X) . Then we can construct some partial solution $\underline{\sigma} \in \mathbb{D}'$ of \mathcal{C} relative to $(\tau, X \downarrow)$ with $\underline{\sigma} \sqsubseteq \sigma$.*

The variable assignment $\underline{\sigma}$ is given by

$$\underline{\sigma} x = \bigsqcap \{ \sigma x' \mid x' \in X, x \leq x' \}.$$

Since we are only interested in as small invariants and solutions as possible, this means that we may restrict ourselves to monotonic variable assignments with downward closed domains. Let $\mathcal{C} \downarrow$ denote the constraint system obtained from \mathcal{C} by replacing every constraint $x \leftarrow f$ with $x \leftarrow f'$ where for $(d, \eta, c) = f(\sigma, \tau)$, $f'(\sigma, \tau) = (d, \eta, c \downarrow)$, and $D_{f'}(\sigma, \tau) = (D_f(\sigma, \tau)) \downarrow$. Thus, the new constraint system differs from the original one in that all sets of spawned variables as well as all occurring sets of variables on which a constraint may depend are downward closed. Let then $\bar{E}_{\mathcal{C}, I}$ denote the constraint system

$$(\sigma, \tau) \sqsupseteq \bar{F}(\sigma, \tau)$$

over $\mathbb{D}' \times \mathbb{D}_2$, where the new right-hand side \bar{F} is the function which first applies the right-hand side F' of $E_{\mathcal{C} \downarrow, I \downarrow}$ and returns (σ'', τ') where, given that F' returns (σ', τ') , the new variable assignment σ'' is defined by

$$\sigma'' x = \begin{cases} \perp & \text{if } \sigma' x' = \perp \\ \bigsqcap \{ \sigma' x' \mid x' \leq x \} & \text{otherwise.} \end{cases}$$

Note that by definition, σ'' is monotonic and has a downward closed domain. We observe:

Proposition 4. *Assume \mathcal{C} is weakly monotonic. Then the following holds:*

1. $\mathcal{C} \downarrow$ is weakly monotonic.

2. The following three statements are equivalent for $\sigma \in \mathbb{D}'$ with domain X :
 - (a) σ is a partial solution of \mathcal{C} relative to (τ, X) ;
 - (b) σ is a partial solution of $\mathcal{C}\downarrow$ relative to (τ, X) ;
 - (c) (σ, τ) is a solution of $\bar{E}_{\mathcal{C}, I}$.
3. The right-hand side of $\bar{E}_{\mathcal{C}, I}$ is monotonic on $\mathbb{D}' \times \mathbb{D}_2$.

The main result of this section is:

Theorem 1. *Let \mathcal{C} denote a constraint system over complete lattices $\mathbb{D}_1, \mathbb{D}_2$ with fixpoint variables from V , and assume that \mathcal{C} is weakly monotonic. Then for every $I \subseteq V$ the following holds:*

1. *Given any partial invariant (τ, X) , $I \subseteq X$, there exists a partial solution σ of \mathcal{C} relative to (τ, X) which is least up to equivalence.*
2. *There exists a least partial invariant (τ, X) for \mathcal{C} with $I \subseteq X$ and X downward closed.*
3. *If V is finite and $\mathbb{D}_1, \mathbb{D}_2$ are of finite height, then the triple (τ, X, σ) consisting of the least partial invariant (τ, X) of \mathcal{C} with $I \subseteq X = X\downarrow$ and a (up to equivalence) least partial solution σ of \mathcal{C} with respect to (τ, X) can be computed through joint fixpoint iteration for $E_{\mathcal{C}, I}$.*

Proof. We only prove assertions 1 and 2. By Propositions 3 and 4, we can prove our main theorem by applying ordinary least fixpoint theory to the constraint system $E = \bar{E}_{\mathcal{C}, I}$. E can be considered as a system of inequalities of the form

$$\begin{aligned}\sigma &\sqsupseteq F_1(\sigma, \tau), \\ \tau &\sqsupseteq F_2(\sigma, \tau).\end{aligned}$$

Consider a partial invariant (τ, X) where X is downward closed. Since \mathbb{D}' is a complete lattice and F_1 is monotonic, standard fixpoint theory guarantees that there is a least σ' in \mathbb{D}' satisfying

$$\sigma' \sqsupseteq F_1(\sigma', \tau) \sqcup \sigma_X,$$

where σ_X returns \perp for every $x \in X$ and \perp otherwise. Since (τ, X) is a partial invariant, we have $\tau \sqsupseteq F_2(\sigma', \tau)$. Hence σ' is a partial solution relative to (τ, X) . On the other hand, every other partial solution $\sigma \in \mathbb{D}'$ of \mathcal{C} is also a solution of the above inequality. Therefore $\sigma' \sqsubseteq \sigma$, implying statement 1 of the theorem.

Now consider the second assertion. Since the set $\mathbb{D}' \times \mathbb{D}_2$ forms a complete lattice (with respect to the componentwise ordering) and the right-hand side of E is monotonic on $\mathbb{D}' \times \mathbb{D}_2$, E has a unique least solution (σ_0, τ_0) . We claim that (τ_0, X) is the least partial invariant for $\mathcal{C}\downarrow$ (and hence also for \mathcal{C}), where X is given by the domain of σ . Note that, by construction, X is downward closed. Consider any other partial invariant (τ_1, X_1) , where X_1 is downward closed. It follows that there is some $\sigma_1 \in \mathbb{D}'$ which is a partial solution of \mathcal{C} relative to (τ_1, X_1) . By construction, (σ_1, τ_1) is also a solution of E . By minimality of (σ_0, τ_0) , we conclude that $\tau_0 \sqsubseteq \tau_1$ and also $X \subseteq X_1$, proving minimality of (τ_0, X) . This completes the proof. \square

In order to apply the results of this section to our analysis framework, we observe that every partial invariant (τ, X) of the constraint system constructed for a given program (relative to an initial set I consisting of all initial calls to main) consists of a safe approximation τ to all possibly occurring global states and a safe super-set X of all reachable program configurations. Since the constraint system is weakly monotonic (given that all behavioural functions are monotonic), there is a least partial invariant which under some finiteness assumptions can be computed effectively.

4. SOLVING SIDE-EFFECTING CONSTRAINT SYSTEMS

Instead of presenting a new algorithm for computing partial invariants, we prefer here to explain how off-the-shelf local solvers for ordinary constraint systems can be customized to serve our needs. Assume we are given a local solver for an *ordinary* constraint system. Such a constraint system is given as a set \mathcal{C}_0 of constraints $x \leftarrow f$, where the right-hand sides are of type

$$f : (V \rightarrow \mathbb{D}_1) \rightarrow \mathbb{D}_1.$$

Following [8], we call an algorithm a (local) solver for (ordinary) constraint systems if it realizes a function Φ which, given a constraint system \mathcal{C}_0 together with a set I of initially reachable variables, returns an *I-stable* partial solution of \mathcal{C}_0 , i.e., a partial variable assignment σ such that

- σx is defined for every $x \in I$;
- if σ is defined for x and $x \leftarrow f$ is a constraint in \mathcal{C}_0 , then σ is defined for all variables accessed during the evaluation of f on σ , and $\sigma x \sqsupseteq f \sigma$.

Such solvers are, e.g., studied in [7,8,11]. For computing a partial invariant for a side-effecting constraint system \mathcal{C} and a given set I of initially reachable variables, we proceed as follows (see Fig. 4). We compute an increasing sequence of approximations to a partial invariant. We start this sequence with the global state $\perp \in \mathbb{D}_2$. From an approximation τ to the partial invariant and the side-effecting system \mathcal{C} , we construct an *ordinary* constraint system (without side effects). This constraint system is obtained from \mathcal{C} by partial application to the given τ . Additionally, every constraint is instrumented in such a way that

1. the set of spawned variables is added to the set of variables accessed by this constraint (in order to trigger their evaluation);
2. the impacts on the global state are monitored. If these impacts are not covered by the current τ , we increase τ .

This instrumented constraint system is then solved by Φ . If no change to the global state has occurred during evaluation of Φ , the iteration terminates. Otherwise, another round with the modified global state is triggered. In this algorithm, whenever the global state is modified, the complete constraint system is scheduled for re-evaluation. Often, however, the global state is of the form

```

fun wrap( $f, \sigma$ )
begin
  let  $\langle d, \eta, s \rangle = f(\sigma, \tau)$  in
    if  $\eta \not\sqsubseteq \tau$  then
       $stable := \text{false};$ 
    fi;
     $\tau := \tau \sqcup \eta;$ 
    foreach  $v \in s$  do  $\sigma(v)$  od;
    return  $d;$ 
end;

begin
   $\tau := \perp;$ 
  repeat
     $stable := \text{true};$ 
     $\mathcal{C}_0 := \{x \leftarrow \lambda\sigma. wrap(f, \sigma) \mid$ 
       $x \leftarrow f \in \mathcal{C}\};$ 
     $\sigma := \Phi(\mathcal{C}_0, I);$ 
  until  $stable$ 
end

```

Fig. 4. The solving scheme.

$\mathbb{D}_2 = G \rightarrow \mathbb{D}$, where G is a finite set of global variables (and abstract heap locations). Thus, the global state does not (explicitly) track relations between globals but captures the abstract properties of different globals independently. If the evaluation of a right-hand side leads to a global state which differs from the original one on the values of just few globals, then re-evaluation of all right-hand sides is overly pessimistic. Instead, it is safe to initiate re-evaluation just of those right-hand sides which depend on the affected globals.

As a corollary of Proposition 2, it is safe for the behavioural functions and, accordingly, also for the constraints to return only that part of the global state which has been modified. This “difference” can be represented by a set of pairs $\langle z, a \rangle$, where $z \in G$ is a modified global and a is its new abstract value. Furthermore, we record for every global z , the set of all constraints accessing z in order to trigger their re-evaluation whenever the value of z changes. For completeness, a corresponding extension of the fixpoint solver from [8] together with an example run are presented in Appendices A and B.

5. EXPERIMENTAL EVALUATION

On the basis of the interprocedural framework from Section 2 and an instance of the solver from Section 4, we have implemented a program analyser generator for C programs with pthreads. The implementation was done in Standard ML using the ckit [12] as our C frontend. We used the framework to implement various analyses for the detection of data-races. For efficiency reasons we organized the analysis as a multi-stage procedure. In the first stage, we determine approximate data values for all globals. This first invariant is then used in the following stages that also track acquired mutex locks.

The implemented analyses handle most of the Posix threads library interface. It must be emphasized that our example analysis from Section 2 is only correct under the assumption that mutex locking and unlocking never fails, which is not necessarily true for the corresponding pthread functions [13]. The locking of mutexes, e.g., may fail due to external interrupts or because the lock already is held by the current thread. Therefore, only in an environment which guarantees

the absence of interrupts, the nonfailure assumption is justified, given that potential repetitions of locks are flagged as errors. For the case where interrupts cannot be excluded, we have generated another variant of the analysis which does not rely on the non-failure assumption for locks but extracts certainty about locking only from checks on the return values of the calls to `pthread_mutex_lock()`.

In the practical experiments reported below, the analysis DR1 assumes that mutex locks always succeed, while the analysis DR2 does not make this assumption.

In collaboration with AIRBUS FRANCE, the implemented analysers were tested on preliminary versions of a large (nonsafety critical) on-board program. The whole system consists of seven components ranging in size from 23 000 to 80 000 LOC (before pre-processing and excluding header files). The analysis was performed on a 1 GigaHertz Athlon with 1 GB memory using SUSE Linux and the SML compiler `smlnj-110.0.7`. Some characteristics of our experiments are reported in Table 1. The first column of the table enumerates the analysed software components, the next three columns provide code statistics (lines of C code, number of threads and mutexes, respectively). Then we report the numbers for the three stages of our analysis. In all three cases, we report the time (in minutes and seconds) and the number of evaluated constraints. In addition, we list for the last two stages the number of reported warnings of potential data-races.

The analysis times for these components varied from a few minutes to less than half an hour. The numbers of flagged potential data-race errors (at most 38 for bench 5) is small enough to be manually inspected by humans. The analysis could not be performed for the component bench 7 as this benchmark uses arrays of threads and mutexes – which is not yet supported by our analyser. Since our analyses compute *safe supersets* of data-race errors, these experiments clearly indicate the high quality of the analysed software. They also demonstrate that the global invariant approach is sufficiently efficient to deal with real-world software components and still precise enough to flag relatively few alarms.

Table 1. Experimental results

| No. | Bench | | | Base | | DR1 | | | DR2 | | |
|-----|--------|----|-----|-----------|---------|----------|---------|-----|-----------|---------|-----|
| | LOC | th | mtx | time | constr | time | constr | wrn | time | constr | wrn |
| 1 | 23 704 | 4 | 14 | 37 s | 85 190 | 37 s | 86 313 | 8 | 41 s | 91 022 | 9 |
| 2 | 31 658 | 5 | 7 | 1 m 00 s | 134 821 | 59 s | 151 052 | 28 | 1 m 22 s | 186 966 | 36 |
| 3 | 37 047 | 1 | 7 | 1 m 17 s | 205 148 | 20 s | 82 679 | 0 | 25 s | 93 053 | 0 |
| 4 | 31 870 | 4 | 8 | 16 s | 55 078 | 20 s | 63 677 | 7 | 21 s | 64 136 | 7 |
| 5 | 29 554 | 4 | 10 | 1 m 37 s | 174 743 | 2 m 56 s | 291 278 | 31 | 15 m 49 s | 716 220 | 38 |
| 6 | 45 660 | 8 | 13 | 17 m 07 s | 260 644 | 1 m 43 s | 223 032 | 6 | 5 m 39 s | 509 274 | 20 |
| 7 | 79 474 | 26 | ? | 4 m 20 s | 463 437 | | ? | | | ? | |

6. RELATED WORK AND FUTURE DIRECTIONS OF RESEARCH

A good overview of the state of the art in the analysis of multi-threaded programs can be found in [14]. Quite a few analyses have recently been developed for optimizing synchronizations in Java [15–18]. In [19], Whaley and Rinard present a combined analysis of pointers and multi-threading for Java. That analysis, though, considers *every access* to a global as potentially harmful. In [20], Jagannathan and Weeks analyse parallel higher-order functional languages but merge all calling contexts to the same function into one.

Cousot [21] considers systems consisting of a finite number of threads, each having a finite number of program points, and concentrates on exhaustive search through all (possibly exponentially many) parallel program configurations. His framework is not (easily) applicable to systems with dynamic creation of threads and recursive procedures. Another approach is to take classical dataflow analysis and extend it to multi-threaded programs by enriching the original control-flow graph with further edges corresponding to interactions of threads which possibly might run in parallel. This approach is proposed by Rugina and Rinard [22] for the construction of a context-sensitive pointer analysis for Cilk. It is also advocated by Naumovich et al. in [23]. Based on a *possibly runs in parallel* analysis [24], they construct an enriched control-flow graph for Java on top of which they model the dynamic thread behaviour via *property automata*. The practicality of such an approach relies on the “density” of the enriched control-flow graph. Recently, Yahav [25] has presented a framework for analysing properties of multi-threaded Java programs based on three-valued logic. His analyses are both extremely precise and extremely inefficient. At least currently, it is open whether his framework can be scaled up to medium-sized programs as well.

The idea of using global invariants for the analysis of multi-threaded systems has also been advocated by Flanagan et al. [26,27]. Their approach is based on an assume-guarantee decomposition, where the invariant is provided by user annotations. They rely on automatic theorem proving and do not provide means of inferring such an invariant.

Presently in our implementation, we assume that all global data potentially can be accessed and modified by all threads which have already been created. Sometimes, this is overly conservative. Salcianu and Rinard [28] introduce the concept of *parallel interaction graphs* (for Java) in order to determine which threads and which functions may have access to certain shared data and thus, potentially, obtain more precise information about globals. It remains for future work to integrate a corresponding analysis into our approach.

ACKNOWLEDGEMENTS

This work was supported by the RTD project IST-1999-20527 “DAEDALUS” of the European FP5 programme. The second author was also partially supported by the Estonian Science Foundation (grant No. 5279).

THE FIXPOINT ALGORITHM

We present the abstract version of the solving algorithm we have implemented in Fig. 5. The algorithm is an extension of (a constraint version of) the worklist solver from [8]. It explores the variable space in a demand-driven fashion: Whenever during evaluation of a constraint (x, f) the value of a variable y is accessed, we beforehand try to compute as good a value for y as possible. Furthermore, we dynamically record that the value of y may influence the constraint (x, f) . If such constraint evaluation has modified the current value for the left-hand side x , we trigger re-evaluation of all constraints we have recorded as possibly influenced by x .

The algorithm proceeds as follows. The set of variables yet to be evaluated is kept in data structure $X : 2^V$. It is initialized with the set I of fixpoint variables in which we are interested. The mappings $\sigma : V \rightarrow \mathbb{D}_1$ and $\tau : G \rightarrow \mathbb{D}$ return the current values of fixpoint variables and globals, respectively. Initially, we assume that σ is undefined everywhere, and $\tau(z)$ equals the least element of \mathbb{D} for every global $z : \mathbb{D}$. The mappings $infl_1 : V \rightarrow 2^{V \times R}$ and $infl_2 : G \rightarrow 2^{V \times R}$ (where $R = ((V \rightarrow \mathbb{D}_1) \times (G \rightarrow \mathbb{D}) \rightarrow \mathbb{D}_1 \times 2^V \times (G \rightarrow \mathbb{D}))$) is the type of right-hand sides) represent the (dynamically tracked) variable dependences. More precisely, $infl_1(x)$ returns the list of all constraints during whose evaluation the value of x has been accessed (analogously for $infl_2$). Initially, both mappings are empty. The mapping $todo : V \rightarrow R$ returns for every variable x , the list of right-hand sides which still are to be (re-) evaluated for this variable. Initially, $todo(x)$ collects *all* right-hand sides of constraints in \mathcal{C} for x . During fixpoint computation, these lists may be removed or re-installed. Finally, the data structure $unsafe : 2^{V \times R}$ collects constraints whose evaluation has been postponed. Initially, it is empty.

After initialization of global data structures, the algorithm iteratively evaluates all variables in X using the procedure *solve*. As a result, the data-structure *unsafe* may contain constraints whose evaluation has been postponed. The right-hand sides of these constraints are merged into the mapping *todo*, while at the same time the corresponding variables occurring as left-hand sides are collected to X . If X is empty (i.e. no constraints had been postponed), the fixpoint iteration is completed. Otherwise, the algorithm proceeds with a next iteration.

The main work of the algorithm is done by the procedure *solve*. When called with a variable x , it first checks whether x has been already considered, and if not, then $\sigma(x)$ and $infl_1(x)$ are initialized. Then all constraints for x , which are currently scheduled for (re-)evaluation, are extracted from the data-structure *todo* and are evaluated. Note that if the constraint (x, f) depends on some fixpoint variable y , this dependency is stored by the function $eval_1$ into $infl_1(y)$. In addition, the variable y is recursively evaluated by *solve*. Similarly, if the constraint depends on some global z , then the dependency is stored by the function $eval_2$ into $infl_2(z)$.

```

proc solve( $x : V$ )
begin
  if  $x \notin \text{dom}(\sigma)$  then  $\sigma(x) := \perp$ ;
   $\text{infl}_1(x) := \emptyset$  fi;
   $W := \text{todo}(x)$ ;  $\text{todo}(x) := \emptyset$ ;
   $\text{new} := \sigma(x)$ ;
  foreach  $f \in W$  do
    let  $(d, \eta, s) = f(\lambda y. \text{eval}_1((x, f), y),$ 
       $\lambda z. \text{eval}_2((x, f), z))$  in
    foreach  $z \in G$  where  $\eta(z) \neq \perp$  do
      if  $\tau(z) \neq \tau(z) \sqcup \eta(z)$  then
         $\tau(z) := \tau(z) \sqcup \eta(z)$ ;
         $\text{unsafe} := \text{unsafe} \cup \text{infl}_2(z)$ ;
         $\text{infl}_2(z) := \emptyset$ ;
      fi;
    od;
    foreach  $y \in s$  do solve( $y$ ) od;
     $\text{new} := \text{new} \sqcup d$ ;
  end;
od;
if  $\sigma(x) \neq \text{new}$  then
   $\sigma(x) := \text{new}$ ;  $U := \emptyset$ ;
  foreach  $(y, f) \in \text{infl}_1(x)$  do
     $\text{todo}(y) := \text{todo}(y) \cup \{f\}$ ;
     $U := U \cup \{y\}$ 
  od;
   $\text{infl}_1(x) := \emptyset$ ;
  foreach  $y \in U$  do solve( $y$ ) od;
fi;
end;

fun eval1( $c : V \times R, y : V$ ) :  $\mathbb{D}_1$ 
begin
  solve( $y$ );  $\text{infl}_1(y) := \text{infl}_1(y) \cup \{c\}$ ;
  return  $\sigma(y)$ 
end;

fun eval2( $c : V \times R, z : G$ ) :  $\mathbb{D}$ 
begin
   $\text{infl}_2(z) := \text{infl}_2(z) \cup \{c\}$ ;
  return  $\tau(z)$ 
end;

begin
   $X := I$ ;  $\sigma := \emptyset$ ;  $\tau := \perp$ ;
   $\text{infl}_1 := \emptyset$ ;  $\text{infl}_2 := \emptyset$ ;
   $\text{todo} := \mathcal{C}$ ;  $\text{unsafe} := \emptyset$ ;
  while  $X \neq \emptyset$  do
    foreach  $x \in X$  do solve( $x$ ) od;
     $X := \emptyset$ ;
    foreach  $(y, f) \in \text{unsafe}$  do
       $\text{todo}(y) := \text{todo}(y) \cup \{f\}$ ;
       $X := X \cup \{y\}$ 
    od;
     $\text{unsafe} := \emptyset$ ;
  end;
end

```

Fig. 5. The solving algorithm.

Evaluating each of these constraints results in a contribution $d : \mathbb{D}_1$ to a value of x together with a set $s : 2^V$ of new variables to be considered (i.e. spawned threads) and a side-effect to globals $\eta : G \rightarrow \mathbb{D}$. For every global z whose value $\tau(z)$ is modified by the side-effect, we move all constraints which depend on z from $\text{infl}_2(z)$ into unsafe for later re-evaluation. Then, all variables in s are evaluated recursively by *solve* and the contribution d is accumulated to the new value $\text{new} : \mathbb{D}_1$ of x .

If the accumulated value new for x is the same as the old one, we are done and the procedure *solve* exits. Otherwise, the value $\sigma(x)$ is updated. Moreover, we must re-evaluate all other constraints whose evaluation has accessed x . These constraints have been accumulated in the entry $\text{infl}_1(x)$. We remove these constraints from $\text{infl}_1(x)$ and merge their right-hand sides with the mapping *todo*. Then the variables corresponding to left-hand sides of these constraints are recursively evaluated by *solve*.

EXAMPLE RUN OF THE SOLVER

As an illustration, consider the C program from Section 2. The corresponding control-flow graphs are depicted in Fig. 2.

Initially, all fixpoint variables in V and all globals have the value \perp . Then fixpoint iteration starts with variable $\langle \text{main}, a \rangle$, where $a = \langle \emptyset, \text{false}, [\text{id} \mapsto \top] \rangle$, meaning that initially the set of held mutex locks is empty, no threads have been created yet, and the local id of main has an unknown value.

In order to determine the result value of the call, the solver tries to evaluate the variable $\langle 3, a \rangle$ which in turn successively demands the evaluations of $\langle 2, a \rangle$, $\langle 1, a \rangle$, and $\langle 0, a \rangle$. Eventually, the constraint for $\langle 0, a \rangle$ is evaluated without further descent into solving other variables. The single constraint for $\langle 0, a \rangle$ returns $(a, \emptyset, \emptyset)$. Given this value, the solver proceeds to the evaluation of the constraint for $\langle 1, a \rangle$ which returns the triple $(a, [z \mapsto \langle \perp, 0 \rangle], \emptyset)$. Thus, the variables $\langle 1, a \rangle$ and z receive the values a and $\langle \perp, 0 \rangle$, respectively. Here, \perp represents the bottom element in the lattice of definitely held mutex locks, i.e., the universal set of mutex locks.

The solver proceeds with the evaluation of the constraint for $\langle 2, a \rangle$. Here, we have to take the effect of the create edge into account. The corresponding constraint returns the triple $(a_1, \emptyset, \{\langle \text{inc}, b \rangle\})$, where $a_1 = \langle \perp, \text{true}, [\text{id} \mapsto \top] \rangle$ represents the new local state of main, i.e., the value for $\langle 2, a \rangle$, and $b = \langle \emptyset, \text{true}, [\text{me} \mapsto \{\&A\}] \rangle$ is the new local state of the newly created thread.

Before ascending to the variable $\langle 3, a \rangle$, the solver triggers the evaluation of the variable $\langle \text{inc}, b \rangle$. Thus, it descends into solving the variables $\langle 7, b \rangle$ down to $\langle 4, b \rangle$, respectively. As 4 is the entry point of the procedure inc, the variable $\langle 4, b \rangle$ receives the value b . Proceeding to the variable $\langle 5, b \rangle$, the solver executes the locking of the mutex at address $\&A$. This results in the new local state $b_1 = \langle \{\&A\}, \text{true}, [\text{me} \mapsto \{\&A\}] \rangle$. Now, the solver executes the constraint corresponding to the edge (5, 6). This edge modifies the global z through the side effect $[z \mapsto \langle \{\&A\}, 1 \rangle]$. The new abstract value for z is

$$\langle \perp, 0 \rangle \sqcup \langle \{\&A\}, 1 \rangle = \langle \{\&A\}, \top \rangle.$$

Since the value of z has been changed, all constraints which depend on it (i.e. the edge (5, 6)) have to be marked as “unsafe” and collected for re-evaluation during next iteration. The local state for $\langle 6, b \rangle$ remains b_1 . Finally, the solver ascends to the variable $\langle 7, b \rangle$ where the unlock is performed yielding the local state $b_2 = \langle \emptyset, \text{true}, [\text{me} \mapsto \{\&A\}] \rangle = b$. Finally, the solver ascends to the variable $\langle \text{inc}, b \rangle$ for which it returns b .

Having finished this detour, the solver continues with the analysis of main by evaluating the constraint for $\langle 3, a \rangle$, which is another call of function inc with the same abstract actual parameter b . Looking up the value of $\langle \text{inc}, b \rangle$ returns the value b . Combining this value with the local state a_1 before the call results in the same

value a_1 . This value is then also returned for the call $\langle \text{main}, a \rangle$, which completes the first round of iteration.

In the next round, all collected “unsafe” constraints are scheduled for re-evaluation. In our case, this is just the constraint corresponding to the edge $(5, 6)$. Calling the solver for $\langle 6, b \rangle$ re-evaluates this constraint, resulting in no new value for $\langle 6, b \rangle$ but in the side-effect $[z \mapsto \langle \{\&A\}, \top \rangle]$. As this is already the current value of z , the fixpoint computation terminates.

In our example, we find that all accesses to the global z which happen after thread creation are protected by the same mutex A , although the function `inc` in principle also could have been called with the address of B as actual parameter. So, only a small finite portion of the potentially quite large constraint system has actually been explored. Only this partial exploration allowed us to ensure that it is always the mutex A which is locked when the variable z is incremented. In fact, this also makes our fixpoint engine reasonably efficient and enables us often to have termination, even in the presence of infinite value domains and recursion, where the functional approach in general is no longer guaranteed to terminate.

REFERENCES

1. Knoop, J., Steffen, B. and Vollmer, J. Parallelism for free: efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Languages and Systems*, 1996, **18**, 268–299.
2. Knoop, J. Parallel constant propagation. In *Proc. of 4th Int. European Conf. on Parallel Processing, Euro-Par '98 (Southampton, Sept. 1998)* (Pritchard, D. and Reeve, J., eds.). LNCS, 1998, **1470**, 445–455.
3. Seidl, H. and Steffen, B. Constraint-based inter-procedural analysis of parallel programs. *Nord. J. Comput.*, 2000, **7**, 375–400.
4. Müller-Olm, M. and Seidl, H. On optimal slicing of parallel programs. In *Proc. of 33rd ACM Symp. on Theory of Computing, STOC'01 (Heraklion, July 2001)*. ACM Press, New York, 2001, 647–656.
5. Fecht, C. GENA – a tool for generating Prolog analyzers from specifications. In *Proc. of 2nd Int. Symp. on Static Analysis, SAS'95 (Glasgow, Sept. 1995)* (Mycroft, A., ed.). LNCS, 1995, **983**, 418–419.
6. Fecht, C. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung*. PhD thesis, Universität des Saarlandes, Saarbrücken, 1997.
7. Fecht, C. and Seidl, H. Propagating differences: an efficient new fixpoint algorithm for distributive constraint systems. *Nord. J. Comput.*, 1998, **5**, 304–329.
8. Fecht, C. and Seidl, H. A faster solver for general systems of equations. *Sci. Comput. Program.*, 1999, **35**, 137–161.
9. Seidl, H. and Fecht, C. Interprocedural analyses: a comparison. *J. Logic Program.*, 2000, **43**, 123–156.
10. Sharir, M. and Pnueli, A. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications* (Muchnick, S. S. and Jones, N. D., eds.). Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981, 189–234.
11. Charlier, B. L. and Hentenryck, P. V. A Universal Top-Down Fixpoint Algorithm. Technical Report CS-92-25, Brown University, Providence, RI, 1992.
12. Heintze, N., Oliva, D. and MacQueen, D. ckit 1.0, March 2000. Available from: <http://www.smlnj.org/doc/ckit/>.

13. Butenhof, D. *Programming with POSIX Threads*. Addison-Wesley, 1997.
14. Rinard, M. Analysis of Multithreaded Programs. In *Proc. of 8th Int. Symp. on Static Analysis, SAS 2001 (Paris, June 2001)* (Cousot, P., ed.). LNCS, 2001, **2126**, 1–19.
15. Diniz, P. and Rinard, M. Lock coarsening: eliminating lock overhead in automatically parallelized object-based programs. *J. Parallel Distrib. Comput.*, 1998, **49**, 218–244.
16. Bogda, J. and Hoelzle, U. Removing unnecessary synchronization in Java. In *Proc. of 14th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'99 (Denver, Colo., Nov. 1999)*. SIGPLAN Notices, 1999, **34**, 10, 35–46.
17. Aldrich, J., Chambers, C., Surer, E. and Eggers, S. Static analysis for eliminating unnecessary synchronization from Java programs. In *Proc. of 6th Int. Symp. on Static Analysis, SAS'99 (Venice, Sept. 1999)* (Cortesi, A. and Filé, G., eds.). LNCS, 1999, **1694**, 19–38.
18. Ruf, E. Effective synchronization removal for Java. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI'00 (Vancouver, BC, June 2000)*. SIGPLAN Notices, 2000, **35**(5), 208–218.
19. Whaley, J. and Rinard, M. Compositional pointer and escape analysis for Java. In *Proc. of 14th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'99 (Denver, Colo., Nov. 1999)*. SIGPLAN Notices, 1999, **34**, 10, 187–206.
20. Jagannathan, S. and Weeks, S. Analyzing stores and references in a parallel symbolic language. In *Proc. of 1994 ACM Conf. on LISP and Functional Programming, LFP'94 (Orlando, Fla., June 1994)*. ACM Press, New York, 1994, 294–305.
21. Cousot, P. Invariance proof methods and analysis techniques for parallel programs. In *Automatic Program Construction Techniques* (Biermann, A., Guiho, G. and Kodratoff, Y., eds.). Collier Macmillan Publishers, London, 1984, 243–271.
22. Rugina, R. and Rinard, M. Pointer analysis for multithreaded programs. In *Proc. of 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI'99 (Atlanta, Ga., May 1999)*. SIGPLAN Notices, 1999, **34**, 5, 77–90.
23. Naumovich, G., Avrunin, G. and Clarke, L. An efficient algorithm for computing MHP information of concurrent Java programs. In *Proc. of 7th European Software Engineering Conf. / 7th ACM SIGSOFT Symp. on Foundations of Software Engineering, ESEC/FSE'99 (Toulouse, Sept. 1999)* (Nierstrasz, O. and Lemoine, M., eds.). LNCS, 1999, **1687**, 338–354.
24. Naumovich, G., Avrunin, G. and Clarke, L. DataFlow analysis for checking properties of concurrent Java programs. In *Proc. of 21th Int. Conf. on Software Engineering, ICSE'99 (Los Angeles, Calif., May 1999)*. ACM Press, New York, 1999, 399–410.
25. Yahav, E. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Conf. Record of 28th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'01 (London, Jan. 2001)*. SIGPLAN Notices, 2001, **36**, 3, 27–40.
26. Flanagan, C., Freund, S. and Qadeer, S. Thread-modular verification for shared-memory programs. In *Proc. of 11th European Symp. on Programming, ESOP 2002 (Grenoble, Apr. 2002)* (Le Métayer, D., ed.). LNCS, 2002, **2305**, 262–277.
27. Flanagan, C., Qadeer, S. and Seshia, S. A modular checker for multithreaded programs. In *Proc. of 14th Int. Conf. on Computer-Aided Verification, CAV 2002 (Copenhagen, July 2002)* (Brinksma, E., Larsen, K. G., eds.). LNCS, 2002, **2404**, 180–194.
28. Salcianu, A. and Rinard, M. Pointer and escape analysis for multithreaded programs. In *Proc. of 8th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPOPP'01 (Snowbird, Utah, June 2001)*. SIGPLAN Notices, 2001, **36**, 7, 12–23.

Globaalinvariandid mitmelõimeliste rakenduste analüüsiks

Helmut Seidl, Varmo Vene ja Markus Müller-Olm

Mitmelõimeliste programmide staatilise analüüsi põhiraskuseks on lõimede paralleelsemantikast tingitud olekuruumi eksponentsiaalne kasv, mistõttu traditsioonilised analüüsimeetodid on nende korral tihti liiga ebaefektiivsed praktikas kasutamiseks. Artiklis on programmide abstraktseks esitamiseks kasutatud uut tüüpi, nn kõrvalefektidega kitsendussüsteeme, kus programmi globaalolek on lõimede lokaalsetest olekutest otseselt eristatud. Sellise kitsendussüsteemi lahendamiseks leitakse kõigepealt globaalolekut abstraktselt kirjeldav minimaalne ohutu globaalinvariant, mille suhtes siis ülejäänud kitsendussüsteem lahendatakse. Tulemusena hoitakse olekuruumi kasv lõimede arvu suhtes lineaarne. On kirjeldatud kõrvalefektidega kitsendussüsteemide formaalseid omadusi ja esitatud baasmeetodid globaalinvariantide arvutamiseks. Samuti on antud ülevaade toodud lähenemise praktilisuse demonstreerimiseks välja töötatud ja realiseeritud suhteliselt efektiivsest analüsaatori prototüübist, mis võimaldab kontrollida kriitiliste ressursside kasutamise korrektsust mitmelõimelistes C-programmides.