Global Invariants for Analyzing Multi-threaded Applications

Helmut Seidl*
Universität Trier, Germany

Varmo Vene†
Tartu University, Estonia

Markus Müller-Olm‡
FernUniversität Hagen, Germany

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. We demonstrate the practicality of this approach by designing and implementing a reasonably efficient flow- and context-sensitive interprocedural data-race analyzer of multi-threaded C.

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 analyzer technology for obtaining sanity checks or even certificates stating the absence of certain programming errors. This type of application does not demand analyzers 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 analyzing larger programs, provided that the number of spurious errors is dramatically decreased. Therefore, we aim at a good balance between precision and analysis time.

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 [16, 15, 25] but for very simple program properties only [17]. In order to arrive at the necessary precision for a non-trivial fragment of C, however, 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 then is used for analyzing each thread individually. Consider, e.g., the control-flow edge shown 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 right in Fig. 1. Traversing the edge conceptually has *two* effects: first, the local state of the edge's target node receives the local variable assignment from the edge's source node, and

*FB IV – Informatik, Universität Trier, D-54286 Trier, Germany; `seidl@psi.uni-trier.de`
†Inst. of Computer Science, Tartu University, Liivi 2, EE-50409 Tartu, Estonia; `varmo@cs.ut.ee`
‡Fachbereich Infomatik, Lehrgebiet Praktische Informatik 5, FernUniversität Hagen, D-58084 Hagen, Germany; on leave from Universität Dortmund; `mmo@ls5.cs.uni-dortmund.de`

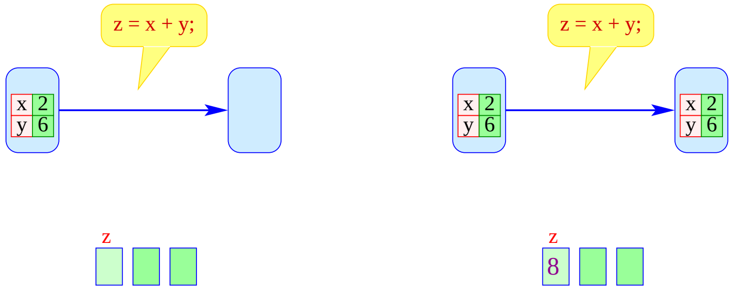


Figure 1: An example control-flow edge and the effect of traversing the edge.

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 presence of multi-threading, this means that we can analyze each thread individually. The separation is particularly successful in applications where the threads are only loosely coupled, i.e., where the controlflow 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 *side-effects* of constraint evaluation.

In earlier work, we have successfully applied general *local* constraint solvers for implementing interprocedural analyses [7, 8, 10, 9, 24]. The architectural idea is to completely separate the fixpoint engine 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 [26] without changing the underlying fixpoint engine. This separation of concerns and sub-division 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 analyzers 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 of 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;                          void main() {
mutex A, B;                             tid id;

void inc(mutex* me) {                   z = 0;
        mutex_lock(me);                 create(&id,inc,&A);
        z = z + 1;                      inc(&A);
        mutex_unlock(me);       }
}
```

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.

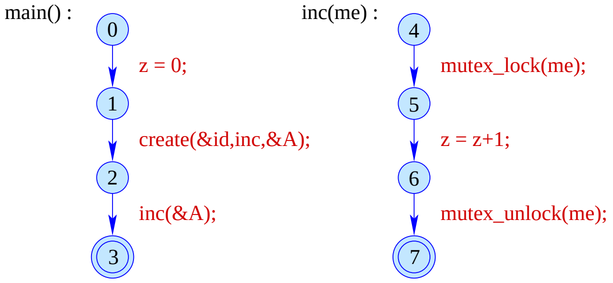


Figure 2: The example control-flow graph.

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 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 $\mathbb{D}_2 = G \to \mathbb{D}$ 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 $\textbf{trans}_e$:

$$\textbf{trans}_e \quad : \quad \mathbb{D}_1 \times \mathbb{D}_2 \to \mathbb{D}_1 \times \mathbb{D}_2$$

The function $\textbf{trans}_e$ 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 $\mathbb{D}_2 = G \to \mathbb{D}$ 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 $\textbf{entry}_e$ and $\textbf{combine}_e$:

$$\begin{array}{lcl} \textbf{entry}_e & : & \mathbb{D}_1 \times \mathbb{D}_2 \to \mathbb{D}_1 \times 2^{\mathcal{F}} \\ \textbf{combine}_e & : & \mathbb{D}_1 \times \mathbb{D}_1 \to \mathbb{D}_1 \end{array}$$

The function $\textbf{entry}_e$ again takes the local 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 onto 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 for this, we let the function $\textbf{entry}_e$ additionally return a set of functions which are potentially called at e. The function $\mathbf{combine}_e$ 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 $\mathbf{create}_e$:

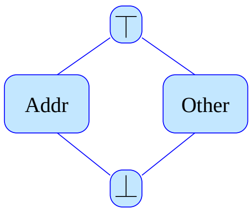
$$\mathbf{create}_e \quad : \quad \mathbb{D}_1 \times \mathbb{D}_2 \to \mathbb{D}_1 \times \mathbb{D}_2 \times \mathbb{D}_1 \times 2^{\mathcal{F}}$$

The function $\mathtt{create}_e$ takes the local 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 $\mathbf{create}_e$ 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 be 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 a 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:

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

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 $\mathbf{trans}_{(0,1)}$ and $\mathbf{trans}_{(5,6)}$ correspond to assignments to the global

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

Figure 3: Behavioral functions for the example program.

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 $\mathbf{trans}_{(4,5)}$ and $\mathbf{trans}_{(6,7)}$ 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 $\rho(\mathsf{me})$ consists of a singleton address set. If $\rho(\mathsf{me})$ consists of more than one address, M is left unchanged. In the case of mutex_unlock(), the abstract value $\rho(\mathsf{me})$ is subtracted from the set M using the auxiliary function rem. In this case, all addresses contained in $\rho(\mathsf{me})$ are removed from M.

Functions $\mathbf{entry}_{(2,3)}$ and $\mathbf{combine}_{(2,3)}$ correspond to the call of the function inc. The function $\mathbf{entry}_{(2,3)}$ returns the initial local state of the call which binds the formal parameter me to the abstract value $\{\&\mathsf{A}\}$. 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 $\mathbf{combine}_{(2,3)}$ ignores these "modifications" and returns the current local state.

Finally, the function $\mathbf{create}_{(1,2)}$ 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(). 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 $\{\&\mathsf{A}\}$. 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 analyzed 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 [26, 24]. Thus, we choose constraints of the form $x \;\Leftarrow\; f$ where $x \in V$ and f is of type:

$$f \quad : \qquad (V \to \mathbb{D}_1) \times \mathbb{D}_2 \to \mathbb{D}_1 \times \mathbb{D}_2 \times 2^V$$

We call constraint systems of this form *side-effecting*. In our application, the right-hand side f takes an assignment $\sigma : V \to \mathbb{D}_1$ 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 behavioral 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,

$$\begin{array}{ll} \langle v, d\rangle \leftarrow \lambda(\sigma, \tau).\ \textbf{let} & (b, \eta) = \textbf{trans}_{(u,v)}\,(\sigma\langle u, d\rangle, \tau) \\ \ \textbf{in} & (b, \eta, \emptyset) \end{array}$$

– If e is a call edge,

$$\begin{array}{ll} \langle v, d\rangle \leftarrow \lambda(\sigma, \tau).\ \textbf{let} & (b_1, F) = \textbf{entry}_{(u,v)}\,(\sigma\langle u, d\rangle, \tau) \\ & b = \bigsqcup\{\textbf{combine}_{(u,v)}\,(\sigma\langle u, d\rangle, \sigma\langle p, b_1\rangle) \mid p \in F\} \\ \ \textbf{in} & (b, \lambda z.\bot, \emptyset) \end{array}$$

– If e is a create edge,

$$\begin{array}{ll} \langle v, d\rangle \leftarrow \lambda(\sigma, \tau).\ \textbf{let} & (b, \eta, b_1, F) = \textbf{create}_{(u,v)}\,(\sigma\langle u, d\rangle, \tau) \\ & s = \{\langle p, b_1\rangle \mid p \in F\} \\ \ \textbf{in} & (b, \eta, s) \end{array}$$

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:

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

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

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

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:

$$f \quad : \qquad (V \to \mathbb{D}_1) \times \mathbb{D}_2 \to \mathbb{D}_1 \times \mathbb{D}_2 \times 2^V$$

We say that the assignment $\sigma : V \to \mathbb{D}_1$ *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 $\sigma : V \to \mathbb{D}_1$ 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 occuring 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 analyzed 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 [8, 10] 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 $\underline{\perp}$ as new least element ("super-bottom"). The *partial variable assignments* from V to $\mathbb{D}_1$ are given as the set of assignments $V \to \underline{\mathbb{D}}_1$. The *domain*, $\mathsf{dom}\,\sigma$, of a partial assignment is given by the set of all variables $x \in V$ with $\sigma\, x \neq \underline{\perp}$.

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:

$$D_f \quad : \quad (V \to \underline{\mathbb{D}}_1) \times \mathbb{D}_2 \to 2^V$$

This function describes a property of the operational behavior 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 "under-specified", i.e., does not provide non-$\underline{\perp}$-values for as many variables as necessary to evaluate f, we return $\underline{\perp}$. Thus, $f\,(\sigma, \tau) = (\underline{\perp}, \perp, \emptyset)$ whenever $\sigma(x) = \underline{\perp}$ 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 $\sigma : V \to \underline{\mathbb{D}}_1$ 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 $\{\langle \mathsf{main}, \top \rangle\}$, and there is a partial invariant (τ, X) such that τ maps the global z to $\langle \{\&\mathsf{A}\}, \top \rangle$. 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 $\underline{\perp}$ otherwise, and the function:

$$F \quad : \quad (V \to \underline{\mathbb{D}}_1) \times \mathbb{D}_2 \to (V \to \underline{\mathbb{D}}_1) \times \mathbb{D}_2$$

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

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

Only those constraints have a non-trivial contribution onto the output (σ_1, τ_1) whose left-hand sides have already a value $\neq \underline{\perp}$. Accordingly, whenever a constraint f contributes to the result, then all variables accessed during this evaluation receive a non-$\underline{\perp}$ 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-$\underline{\perp}$ values for all $x \in I$. Hence, the (trivial) least solution $(\lambda\, x.\underline{\perp}, \perp)$ 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 $E_{\mathcal{C},I}$.* $\square$

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, then we simply can 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

$$\begin{array}{rl} f(\sigma,\tau) = & \textbf{let } (d,\eta,s) = f'(\sigma,\tau) \\ & \textbf{in } (d, \eta \sqcup \tau, s) \end{array}$$

Then the following holds:

Proposition 2 *For any partial variable assignment $\sigma : V \to \underline{\mathbb{D}}_1$ 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 $E_{\mathcal{C}',I}$.* $\square$

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 F has a least solution by the fixpoint theorem of Knaster/Tarski. In particular, this is the case when all functions f and D_f are monotonic (w.r.t. the obvious orderings). As observed in [8, 10], 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., [26] 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 behavioral functions trans_e, entry_e, $\mathsf{combine}_e$ 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: $\langle r_1, a_1\rangle \leq \langle r_2, a_2\rangle$ iff $r_1 = r_2$ 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* w.r.t. 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 non-monotonic assignments to comparable monotonic ones. Some remarks are appropriate here.

- A partial variable assignment σ, i.e., a mapping $\sigma : V \to \underline{\mathbb{D}}_1$, is considered as monotonic iff for all $x_1 \leq x_2$, $\sigma\, x_2 \neq \underline{\perp}$ implies $\sigma\, x_1 \leq \sigma\, x_2$ (i.e., we demand nothing if $\sigma\, x_2 = \underline{\perp}$).
- 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 $\sigma_1, \sigma_2 : V \to \underline{\mathbb{D}}_1$ as *equivalent* if:

  1. for any $x_1 \in V$, there is some $x_2 \in V$ with $x_1 \leq x_2$ s.t. $\sigma_1\, x_1 \sqsubseteq \sigma_2\, x_2$, and also
  2. for any $x_1 \in V$, there is some $x_2 \in V$ with $x_1 \leq x_2$ s.t. $\sigma_2\, x_1 \sqsubseteq \sigma_1\, x_2$.

  If V has finite height, then monotonic assignments are equivalent iff they *agree* on variables which are maximal w.r.t. "$\leq$" 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 $V \to \underline{\mathbb{D}}_1$ 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 $(\tau, X\!\downarrow)$ with $\underline{\sigma} \sqsubseteq \sigma$.* $\square$

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 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 onto 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:

$$\sigma''\, x = \begin{cases} \underline{\perp} & \text{if } \sigma'\, x' = \underline{\perp} \\ \bigsqcup\{\sigma'\, x' \mid x' \leq x\} & \text{otherwise} \end{cases}$$

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 $\mathbb{D}' \times \mathbb{D}_2$.* $\square$

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 $\mathcal{C}$ w.r.t. (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 $E = \bar{E}_{\mathcal{C}\downarrow,I}$. E can be considered as a system of in-equations of the form:

$$\begin{array}{rcl} \sigma & \sqsupseteq & F_1(\sigma, \tau) \\ \tau & \sqsupseteq & F_2(\sigma, \tau) \end{array}$$

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 $\bot$ for every $x \in X$ and $\underline{\bot}$ 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 in-equation. 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 (w.r.t. 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 behavioral 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:

$$f \quad : \quad (V \to \mathbb{D}_1) \to \mathbb{D}_1$$

Following [10], 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

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

begin
  $\tau := \bot$;
  repeat
    stable := **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

Figure 4: The solving scheme.

- $\sigma\, x$ 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 [4, 10, 9]. 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 $\bot \in \mathbb{D}_2$. 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 onto the global state are monitored. If these impacts are not covered by the current τ, we increase τ.

This instrumented constraint system then is 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 reevaluation. Often, however, the global state is of the form $\mathbb{D}_2 = G \to \mathbb{D}$ 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 reevaluation of all right-hand sides is overly pessimistic. Instead, it is safe to initiate reevaluation just of those right-hand sides which depend onto the affected globals.

As a corollary of Proposition 2, it is safe for the behavioral 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 reevaluation whenever the value of z changes. For completeness, a corresponding extension of the fixpoint solver from [10] together with an example run are presented in the Appendices A and B.

5 Experimental Evaluation

Based on the interprocedural framework from Section 2 and an instance of the solver from Section 4, we have implemented a program analyzer generator for C programs with pthreads. The

bench				Base		DR1			DR2		
#	LOC	th	mtx	time	constr	time	constr	wrn	time	constr	wrn
1	23704	4	14	37s	85190	37s	86313	8	41s	91022	9
2	31658	5	7	1m 00s	134821	59s	151052	28	1m 22s	186966	36
3	37047	1	7	1m 17s	205148	20s	82679	0	25s	93053	0
4	31870	4	8	16s	55078	20s	63677	7	21s	64136	7
5	29554	4	10	1m 37s	174743	2m 56s	291278	31	15m 49s	716220	38
6	45660	8	13	17m 07s	260644	1m 43s	223032	6	5m 39s	509274	20
7	79474	26	?	4m 20s	463437		?			?	

Figure 5: Experimental results.

implementation was done in Standard ML using the ckit [13] 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 then is 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 [3]. 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 absence of interrupts, the non-failure 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(). We then leave it to the user which analysis better serves her needs.

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 analyzers were tested on preliminary versions of a large (non-safety 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 Fig. 5. The first column of Table 5 enumerates the analyzed 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 analyzer. Since our analyses compute *safe supersets* of data-race errors, these experiments clearly indicate the high quality of the analyzed 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.

6 Related Work and Future Directions of Research

A good overview on the state of the art in the analysis of multi-threaded programs can be found in [20]. Quite a few analyses have been recently developed for optimizing synchronizations in

Java [6, 2, 1, 21]. In [27], Whaley and Rinard present a combined analysis of pointers and multi-threading for Java. The latter analysis, though, considers *every access* to a global as potentially harmful. In [14], Jagannathan and Weeks analyze parallel higher-order functional languages but merge all calling contexts to the same function into one.

Cousot 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 [5]. 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 controlflow graph with further edges corresponding to interactions of threads which possibly might run in parallel. This approach is proposed by Rugina and Rinard for the construction of a context-sensitive pointer analysis for Cilk [22]. It is also advocated by Naumovich et al. in [18]. Based on a *possibly runs in parallel* analysis [19], they construct an enriched control-flow graph for Java on top of which they model the dynamic thread behavior via *property automata*. The practicality of such an approach relies on the "density" of the enriched control-flow graph. Recently, Yahav has presented a framework for analyzing properties of multi-threaded Java programs based on three-valued logic [28]. His analyses are both extremely precise and extremely inefficient. At least currently, it is open whether it 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. in [11, 12]. 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 to infer 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 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 [23]. It remains for future work to integrate a corresponding analysis into our approach.

7 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.

References

[1] J. Aldrich, C. Chambers, E. Sirer, and S. Eggers. Static Analysis for Eliminating Unnecessary Synchronization from Java Programs. In *5th Int. Static Analysis Symposium (SAS)*, pages 19–38. LNCS 1694, Springer Verlag, 1999.

[2] J. Bogda and U. Hoelzle. Removing Unnecessary Synchronization in Java. In *14th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 35–46. SIGPLAN Notices 34(10), 1999.

[3] D. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[4] B. L. Charlier and P. V. Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report CS-92-25, Brown University, Providence, RI 02912, 1992.

[5] P. Cousot. Invariance Proof Methods and Analysis Techniques for Parallel Programs. In A. Biermann, G. Guiho, and Y. Kodratroff, editors, *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Collier Macmillan Publishers, London, 1984.

[6] P. Diniz and M. Rinard. Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-Based Programs. *Journal of Parallel and Distributed Computing*, 49(2):218–244, 1998.

[7] C. Fecht. GENA - A Tool for Generating Prolog Analyzers from Specifications. In *2nd Static Analysis Symposium (SAS)*, pages 418–419. LNCS 983, 1995.

[8] C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung*. PhD thesis, Universität des Saarlandes, Saarbrücken, 1997.

[9] C. Fecht and H. Seidl. Propagating Differences: An Efficient New Fixpoint Algorithm for Distributive Constraint Systems. *Nordic Journal of Computing (NJC)*, 5(4):304–329, 1998.

[10] C. Fecht and H. Seidl. A Faster Solver for General Systems of Equations. *Science of Computer Programming (SCP)*, 35(2):137–161, 1999.

[11] C. Flanagan, S. Freund, and S. Qadeer. Thread-Modular Verification for Shared-Memory Programs. In *11th European Symposiym on Programming (ESOP)*, pages 262–277. LNCS 2305, Springer Verlag, 2002.

[12] C. Flanagan, S. Qadeer, and S. Seshia. A Modular Checker for Multithreaded Programs. In *Computer-Aided Verification (CAV)*. LNCS, Springer Verlag, July 2002. to appear.

[13] N. Heintze, D. Oliva, and D. MacQueen. ckit 1.0, March 2000. Available from: `http://plan9.bell-labs-com/cm/what/smlnj/doc/ckit`.

[14] S. Jagannathan and S. Weeks. Analyzing Stores and References in a Parallel Symbolic Language. In *ACM Conf. on LISP and Functional Programming*, pages 294–305, 1994.

[15] J. Knoop. Parallel Constant Propagation. In *4th European Conference on Parallel Processing (Euro-Par)*, volume 1470 of *Lecture Notes in Computer Science (LNCS)*, pages 445–455. Springer-Verlag, 1998.

[16] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, 1996.

[17] M. Müller-Olm and H. Seidl. On Optimal Slicing of Parallel Programs. In *ACM Symposium on Theory of Computing (STOC)*, pages 647–656, 2001.

[18] G. Naumovich, G. Avrunin, and L. Clarke. An Efficient Algorithm for Computing MHP Information of Concurrent Java Programs. In *7th ACM SIGSOFT Symp. on the Foundations of Software Engineering (FSE)*, pages 338–354, 1999.

[19] G. Naumovich, G. Avrunin, and L. Clarke. DataFlow Analysis for Checking Properties of Concurrent Java Programs. In *21th Int. Conf. on Software Engineering (ICSE)*, pages 399–410, 1999.

[20] M. Rinard. Analysis of Multithreaded Programs. In *7th Int. Static Analysis Symposium (SAS)*, pages 1–19. LNCS 2126, Springer Verlag, 2001.

[21] E. Ruf. Effective Synchronization Removal for Java. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 208–218. SIGPLAN Notices 35(5), 2000.

[22] R. Rugina and M. Rinard. Pointer Analysis for Multithreaded Programs. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 77–90. SIGPLAN Notices 34(5), 1999.

[23] A. Salcianu and M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, pages 12–23, 2001.

[24] H. Seidl and C. Fecht. Interprocedural Analyses: A Comparison. *Journal of Logic Programming (JLP)*, 43(2):123–156, 2000.

[25] H. Seidl and B. Steffen. Constraint-Based Inter-Procedural Analysis of Parallel Programs. *Nordic Journal of Computing (NJC)*, 7(4):375–400, 2000.

[26] M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[27] J. Whaley and M. Rinard. Compositional Pointer and Escape Analysis for Java. In *14th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 187–206. SIGPLAN Notices 34(10), 1999.

[28] E. Yahav. Verifying Safety Properties of Concurrent Java Programs Using 3-valued Logic. In *28th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, pages 27–40, 2001.

A The fixpoint algorithm

We present the abstract version of the solving algorithm we have implemented in Fig. 6. The algorithm is an extension of (a constraint version of) the worklist solver from [10]. 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 $\sigma : V \to \mathbb{D}_1$ and $\tau : G \to \mathbb{D}$ 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 $\mathit{infl_1} : V \to 2^{V\times R}$ and $\mathit{infl_2} : G \to 2^{V\times R}$ (where $R = ((V \to \mathbb{D}_1) \times (G \to \mathbb{D}) \to \mathbb{D}_1 \times 2^V \times (G \to \mathbb{D}))$ is the type of right-hand sides) represent the (dynamically tracked) variable dependences. More precisely, $\mathit{infl_1}(x)$ returns the list of all constraints during whose evaluation the value of x has been accessed (analogously for $\mathit{infl_2}$). Initially, both mappings are empty. The mapping $\mathit{todo} : V \to R$ returns for every variable x, the list of right-hand sides which still are to be (re-) evaluated for this variable. Initially, $\mathit{todo}(x)$ 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 $\mathit{unsafe} : 2^{V\times R}$ 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 $\mathit{infl_1}(x)$ 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, then this dependency is stored by the function $\mathit{eval_1}$ into

```
proc solve(x : V)
begin
  if x ∉ dom(σ) then σ(x) := ⊥; infl₁(x) := ∅ fi;
  W := todo(x); todo(x) := ∅; new := σ(x);
  foreach f ∈ W do
    let (d, η, s) = f ( λy. eval₁((x, f), y) ,
                        λz. eval₂((x, f), z) ) in
      foreach z ∈ G where η(z) ≠ ⊥ do
        if τ(z) ≠ τ(z) ⊔ η(z) then
          τ(z) := τ(z) ⊔ η(z);
          unsafe := unsafe ∪ infl₂(z);
          infl₂(z) := ∅;
        fi;
      od;
      foreach y ∈ s do solve(y) od;
      new := new ⊔ d;
    end;
  od;
  if σ(x) ≠ new then
    σ(x) := new; U := ∅;
    foreach (y, f) ∈ infl₁(x) do
      todo(y) := todo(y) ∪ {f};
      U := U ∪ {y}
    od;
    infl₁(x) := ∅;
    foreach y ∈ U do solve(y) od;
  fi;
end;

fun eval₁(c : V × R, y : V) : 𝔻₁
begin
  solve(y); infl₁(y) := infl₁(y) ∪ {c};
  return σ(y)
end;

fun eval₂(c : V × R, z : G) : 𝔻
begin
  infl₂(z) := infl₂(z) ∪ {c};
  return τ(z)
end;

begin
  X := I; σ := ∅; τ := ⊥;
  infl₁ := ∅; infl₂ := ∅;
  todo := 𝒞; unsafe := ∅;
  while X ≠ ∅ do
    foreach x ∈ X do solve(x) od;
    X := ∅;
    foreach (y, f) ∈ unsafe do
      todo(y) := todo(y) ∪ {f};
      X := X ∪ {y}
    od;
    unsafe := ∅;
  end;
end
```

Figure 6: The solving algorithm.

infl₁(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 $infl_2(z)$.

Evaluating each of these constraints result 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) as well a side-effect to globals $\eta : G \to \mathbb{D}$. For every global z whose value $\tau(z)$ is modified by the side-effect, we move all constraints which depend on z from $infl_2(z)$ 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 $new : \mathbb{D}_1$ of x.

If the accumulated value *new* for x is the same as the old one, then we are done and 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 $infl_1(x)$. We remove these constraints from $infl_1(x)$ 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*.

B 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 $\bot$. Then fixpoint iteration starts with variable $\langle \mathsf{main}, a\rangle$ where $a = \langle \emptyset, \mathbf{false}, [\mathsf{id} \mapsto \top]\rangle$ 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 of 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 $(a, [\mathsf{z} \mapsto \langle \bot, 0\rangle], \emptyset)$. Thus, the variables $\langle 1, a\rangle$ and z receive the values a and $\langle \bot, 0\rangle$, respectively. Here, $\bot$ 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 $(a_1, \emptyset, \{\langle \mathsf{inc}, b\rangle\})$ where $a_1 = \langle \bot, \mathbf{true}, [\mathsf{id} \mapsto \top]\rangle$ represents the new local state of main, i.e., the value for $\langle 2, a\rangle$, and $b = \langle \emptyset, \mathbf{true}, [\mathsf{me} \mapsto \{\&\mathsf{A}\}]\rangle$ 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 $\langle \mathsf{inc}, b\rangle$. 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 $b_1 = \langle \{\&\mathsf{A}\}, \mathbf{true}, [\mathsf{me} \mapsto \{\&\mathsf{A}\}]\rangle$. Now, the solver executes the constraint corresponding to the edge $(5, 6)$. This edge modifies the global z through the side effect $[\mathsf{z} \mapsto \langle \{\&\mathsf{A}\}, 1\rangle]$. The new abstract value for z is

$$\langle \bot, 0\rangle \sqcup \langle \{\&\mathsf{A}\}, 1\rangle = \langle \{\&\mathsf{A}\}, \top\rangle$$

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 $b_2 = \langle \emptyset, \mathbf{true}, [\mathsf{me} \mapsto \{\&\mathsf{A}\}]\rangle = b$. Finally, the solver ascends to the variable $\langle \mathsf{inc}, b\rangle$ 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 $\langle \mathsf{inc}, b\rangle$, 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 $\langle \mathsf{main}, a\rangle$ – 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 $[\mathsf{z} \mapsto \langle\{\&\mathsf{A}\}, \top\rangle]$. 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 actually has 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 presence of infinite value domains and recursion where the functional approach in general is no longer guaranteed to terminate.