Interprocedural Invariants

Helmut Seidl[1], Varmo Vene[2], and Markus Müller-Olm[3]

[1] FB IV – Informatik, Universität Trier, D-54286 Trier, Germany, seidl@psi.uni-trier.de
[2] Inst. of Computer Science, Tartu University, Liivi 2, EE-50409 Tartu, Estonia, varmo@cs.ut.ee
[3] FB4, Lehrstuhl für Programmiersysteme, Universität Dortmund, D-44221 Dortmund, Germany, mmo@cs.uni-dortmund.de

Abstract. We propose the concept of *side-effecting* constraint systems where each constraint for a variable x may additionally affect other variables in the system. We present a *local* solver which explores the variable space demand-driven. We explain how side-effected constraint systems together with local constraint solving can be used for the design and implementation of a framework for interprocedural analyses of multi-threaded programs. We demonstrate the practicality of this approach by designing a reasonably efficient controlflow- and context-sensitive interprocedural data race analyzer of multi-threaded C programs.

1 Introduction

The DAEDALUS project is a common 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 to obtain sanity checks or even certificates for 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 analysis time on larger programs - given only that the number of spurious errors is dramatically decreased. Accordingly, we aim at a better 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 [14, 13, 24] but for very simple program properties only [16]. 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 analyzes. Our key observation is the following. 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 from 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 in fig.

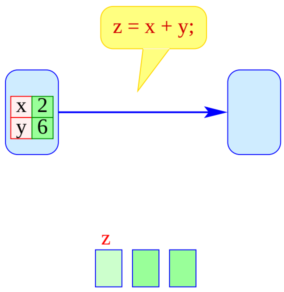


Fig. 1. An example control-flow edge ...

2. 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 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., the controlflow mainly depends on the values of locals. We are left, however, with the task of inferring an as tight approximation to the 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 our earlier work, we have successfully applied general *local* constraint solvers for implementing interprocedural analyses [7, 8, 10, 9, 23]. The architectural idea is to completely separate the fixpoint engine from the program analysis framework — implying 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 length without changing the underlying fixpoint engine. This separation of concerns and sub-division into as small software components as possible 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*. Therefore, we present an application independent formulation of partial invariants over a class of generalized con-

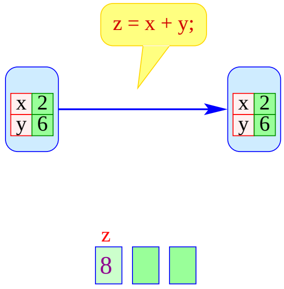


Fig. 2. ... and the effect of traversing the edge.

straint systems which we call *side-effecting*. We give a characterization of partial invariants for side-effecting constraint systems by means of ordinary constraint systems and present a sufficient condition guaranteeing the existence of a unique least partial invariant. We demonstrate the appropriateness of our approach by exhibiting how a framework for concurrent interprocedural analysis can be based on partial invariants. This framework conveniently allows to specify, e.g., data-race analyzers for multi-threaded C. Inferring a least partial invariant, means for the analysis that only those effects onto globals are recorded which occurred during tracking of function calls.

In the second half of the present paper, we show how *local* constraint solving can be generalized to infer partial invariants. Local constraint solving explores the variables of the constraint system *on demand*. Evaluation of variables whose values do not contribute to the fixpoint variables of interest is abandoned. For classical interprocedural analysis, this means that *efficiency* is gained by analyzing only those procedure calls which actually occur during fixpoint evaluation. Tracking of variable dependences only, however, cannot be directly applied to side-effecting constraints systems. The reason, intuitively, is that the values of globals (at least formally) may not depend just on a single instance of a procedure call, but *all* of them — although only those should give a non-trivial contribution which are actually reached. Therefore, we propose an extension of local solving which deals with "side-effects" of constraints in a systematic way. Collecting the side-effects just of reachable procedure calls, does not only increase efficiency but also *precision*.

Summarizing, the contributions of our work are the following:

- As a formal basis, we introduce a new kind of constraint systems and the notion of partial invariants and provide sufficient criteria for the existence and uniqueness of least partial invariants;
- We demonstrate the appropriateness of the new notion by exhibiting an interprocedural framework based on partial invariants;
- We develop a local fixpoint algorithm which is able to compute partial invariants efficiently.

2 Side-Effecting Constraint Systems

We distinguish two kinds of variables: The set V_1 of *fixpoint variables* corresponding to reachable program configurations, and the set V_2 representing all kinds of *globals,* such as global program variables and heap cells. Let $\mathbb{D}_1, \mathbb{D}_2$ denote complete lattices for fixpoint variables and globals, respectively. A side-effecting constraint (over $\mathbb{D}_1, \mathbb{D}_2$, V_1 and V_2) is of the form:

$$x \quad \leftarrow \quad f$$

where $x \in V_1$ and f is of type:

$$f \quad : \quad (V_1 \to \mathbb{D}_1) \times (V_2 \to \mathbb{D}_2) \to \mathbb{D}_1 \times 2^{V_1} \times (V_2 \to \mathbb{D}_2)$$

Thus, f takes a fixpoint variable assignment $\sigma : V_1 \to \mathbb{D}_1$ and an environment $\tau : V_2 \to \mathbb{D}_2$ for the globals and returns a triple (d, s, η) consisting of a return value d for the left-hand side x together with a set s of further variables "of interest" and a variable assignment η representing the necessary assumptions on the environment of globals for solving the constraint. In many cases, though, assumptions are needed upon the values of few globals only. This can be expressed, e.g., by setting $\eta\, z = \bot$ for all other globals z.
Let τ denote an environment from $V_2 \to \mathbb{D}_2$. An assignment $\sigma : V_1 \to \mathbb{D}_1$ satisfies the constraint relative to τ iff for $f\,(\sigma, \tau) = (d, s, \eta)$,

- $d \sqsubseteq \sigma\, x$, and also
- $\eta\, z \sqsubseteq \tau\, z$ for every $z \in V_2$.

In this sense, we can define a *complete* solution of a set $\mathcal{C}$ of constraints relative to a environment τ as a mapping $\sigma : V_1 \to \mathbb{D}_1$ satisfying *all* constraints in $\mathcal{C}$ relative to τ. An environment τ is called an *invariant* of $\mathcal{C}$ if there exists a solution of $\mathcal{C}$ relative to τ.
In our application of interprocedural analysis of multi-threaded code, however, we are not interested in *complete* solutions. The variables from V_1 represent all *formally possible* local program configurations – even those which are in fact unreachable. The invariant, though, is only needed for configurations which are "reachable". Clearly, the smaller the set of program configurations we have to 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. Here, we generalize this approach to side-effecting constraint systems with global invariants.
We introduce the notion of *partial* variable assignments. Let $\underline{\bot}$ denote a new element and $\underline{D}_1$ the complete lattice which we obtain from $\mathbb{D}_1$ by adding $\underline{\bot}$ as new least element ("super-bottom"). The partial variable assignments from V_1

to $\mathbb{D}_1$ then are given as the set of assignments $V_1 \to \underline{\mathbb{D}}_1$. The *domain*, $\mathsf{dom}\,\sigma$, of a partial assignment is given by the set of all variables $x \in V_1$ with $\sigma\,x \neq \underline{\perp}$.
Then 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:

$$D_f \quad : \qquad (V_1 \to \underline{\mathbb{D}}_1) \times (V_2 \to \mathbb{D}_2) \to 2^{V_1}$$

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 straight forward, however, to extract D_f if f is specified in some kind of expression language 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 $D_f\ (\sigma, \tau) = V \subseteq V_1$, then f returns the same value on every pair (σ_1, τ) where $\sigma_1\,x = \sigma\,x$ for $x \in V$ (and $\sigma_1\,x$ arbitrary otherwise).
When applying f to a partial envirnoment 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, $f\,(\sigma, \tau) = (\underline{\perp}, \emptyset, \lambda\,z.\perp)$ whenever $\sigma(x) = \underline{\perp}$ for some $x \in D_f(\sigma, \tau)$.
Let (τ, V) denote a pair consisting of an environment $\tau : V_2 \to \mathbb{D}_2$ and a subset $V \subseteq V_1$. A partial variable assignment $\sigma : V_1 \to \underline{\mathbb{D}}_1$ is called a *partial solution* of the constraint system $\mathcal{C}$ relative to (τ, V) iff for every variable $x \in V$ and every constraint $x \leftarrow f$ for x in $\mathcal{C}$ with $(d, s, \eta) = f\,(\sigma, \tau)$ the following holds:

- $\sigma\,x \neq \underline{\perp}$;
- $d \sqsubseteq \sigma\,x$,
- $s \cup D_f(\sigma, \tau) \subseteq V$, and also
- $\eta \sqsubseteq \tau$.

The pair (τ, V) is called a *partial invariant* of $\mathcal{C}$ if there exists a partial solution of $\mathcal{C}$ relative to (τ, V).
Every constraint system with $V_1 \neq \emptyset$ has at least two partial invariants. The first one holds for an *empty set* of reachable variables only and therefore may map each globat to $\perp$, i.e., is given by: $(\lambda\,z.\perp, \emptyset)$. The second extreme considers *all* fixpoint variables as reachable and provides no information about the globals, i.e., maps each of them to $\top$. It is given by: $(\lambda\,z.\top, V_1)$. Both invariants are not very interesting. The smaller the invariant, though, the more precise an information is provided for the globals. In our applications, we are given a subset $X \subseteq V_1$ of initial configurations which are trivially reachable and therefore must be included into the set of reachable configurations. Thus, the goal is to determine an as small partial invariant (τ, V) as possible such that $X \subseteq V$.
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 $X \subseteq V_1$ of initial variables the corresponding *combined constraint* $E_{\mathcal{C},X}$:

$$(\sigma, \tau) \sqsupseteq F\ (\sigma, \tau) \sqcup (\sigma_X, \lambda\,\_.\perp)$$

Here, $\sigma_X\, x = \perp$ if $x \in X$ and $\underline{\perp}$ otherwise, and the function:

$$F \quad : \quad (V_1 \to \underline{\mathbb{D}}_1) \times (V_2 \to \mathbb{D}_2) \to (V_1 \to \underline{\mathbb{D}}_1) \times (V_2 \to \mathbb{D}_2)$$

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

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

Thus, 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 also all variables accessed during this evaluation receive a non-$\underline{\perp}$ value. Indeed, these two features will allow us to design a local solver which explores the space of fixpoint variables backward through variable dependences.
The function F as defined above, returns non-$\underline{\perp}$ values for all $x \in X$. Hence, the (trivial) least solution $(\lambda\, x.\underline{\perp}, \lambda\, z.\perp)$ is ruled out whenever $X \neq \emptyset$. We have:

Proposition 1. *For an environment $\tau : V_2 \to \mathbb{D}_2$ and a partial variable assignment σ with a domain V exceeding X, the following two statements are equivalent:*

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

Proposition 1 is the key observation which allows us to apply fixpoint methods both to provide sufficient conditions for the existence of unique least partial invariants and for providing algorithmic tools for effectively computing them.
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], interprocedural analyses, however, usually do *not* introduce monotonic constraints. The abstract effect of a procedure p conceptually can be thought of as described by a (monotonic) function. In many cases, however, this function is too complicated to be treated as a whole and therefore is replaced by the *set* of variables $\langle p, a\rangle$, $a \in \mathbb{D}_1$, describing the results of *abstract procedure calls* of p on 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, however, is no longer monotonic — but just *weakly* monotonic. Weak monotonicity is defined relative to a partial ordering on the set of variables V_1. In case of abstract procedure calls, this ordering is given by: $\langle p_1, a_1\rangle \leq \langle p_2, a_2\rangle$ iff $p_1 = p_2$ and $a_1 \sqsubseteq a_2$.
A system $\mathcal{C}$ of constraints (over V_1, V_2, $\mathbb{D}_1$ 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 environments $\tau_i : V_2 \to \mathbb{D}_2$, 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 $x_2 \in V$ 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 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 $\sigma : V_1 \to \mathbb{D}_1$, is considered as monotonic iff for all $x_1 \leq x_2$, $\sigma\, x_2 \neq \bot$ implies $\sigma\, x_1 \leq \sigma\, x_2$ (i.e., we demand nothing in case $\sigma\, x_2 = \bot$, i.e., is undefined).
- The pre-ordering on subsets $X_1, X_2 \subseteq V_1$ 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 $x_2 \in X_2$.
  Instead of using a pre-ordering here, we could also use the subset ordering on the corresponding downward closed subsets of variables. In particular:
- We view two monotonic assignments $\sigma_1, \sigma_2 : V_1 \to \mathbb{D}_1$ as *equivalent* if the following holds:
  1. for every $x_1 \in V_1$, there is some $x_2 \in V_1$ with $x_1 \leq x_2$ such that $\sigma_1\, x_1 \sqsubseteq \sigma_2\, x_2$, and vice versa,
  2. for every $x_1 \in V_1$, there is some $x_2 \in V_1$ with $x_1 \leq x_2$ such that $\sigma_2\, x_1 \sqsubseteq \sigma_1\, x_2$.

  Thus, if V_1 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.

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 variables from V_1 and globals from V_2, and assume that $\mathcal{C}$ is weakly monotonic. Then for every $X \subseteq V_1$ the following holds:*

1. *There exists a least partial invariant (τ, V) for $\mathcal{C}$ with $X \subseteq V$.*
2. *Given any partial invariant (τ, V), $X \subseteq V$, there exists a partial solution σ of $\mathcal{C}$ relative to (τ, V) which is least upto equivalence.*
3. *If V_1 is finite and $\mathbb{D}_1, \mathbb{D}_2$ are of finite height, then the triple (τ, V, σ) consisting of the least partial invariant (τ, V) of $\mathcal{C}$ with $X \subseteq V$ and a (upto equivalence) least partial solution σ of $\mathcal{C}$ w.r.t. (τ, V) can be computed through joint fixpoint iteration for $E_{\mathcal{C},X}$.* □

3 The Multi-threaded Interprocedural Framework

In this section, we show how to build, on top of side-effecting constraint systems and partial invariants as presented in the last section, a framework for interprocedural program analysis which is flexible enough to deal with heap allocated data structures and multi-threading. Each program consists of a finite set $\mathcal{F}$ of functions. As usual, we assume that each function $f \in \mathcal{F}$ is specified via a finite control-flow graph representing the body of the function. Consider, e.g., the following C program:

```
extern void create (tid* id, void* f(void* x), void* x);

extern void mutex_init (mutex* me);
extern void mutex_lock (mutex* me);
extern void mutex_unlock (mutex* me);
```

```
int z;
mutex A, B;

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

```
void main() {
        tid id;
        mutex_init(&A);
        z = 0;
        create(&id,inc,&A);
        inc(&A);
       }
```

Here, the external functions 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. Assuming that they always succeed, we have given them the return type void. For simplicity, we have omitted the necessary type casts for the arguments of create to have the program correctly type-checked. The corresponding control-flow graphs are depicted in fig. 3.

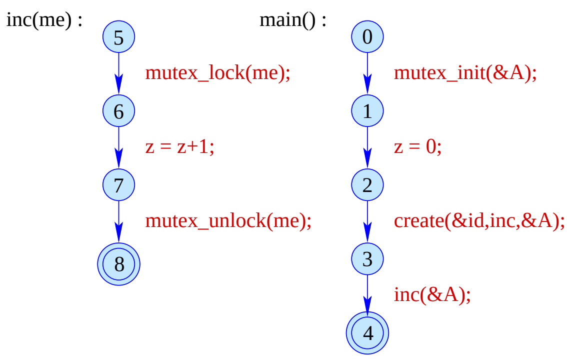


Fig. 3. A simple multi-threaded program.

Assume that the complete lattices of abstract values are given by $\mathbb{D}_1$ (for local program states) and $\mathbb{D}_2$ (for globals), respectively. Then set V_1 of fixpoint variables maintained by the analyzer consists of:

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

In particular, the partial ordering on these variables is given by $\langle f_1, d_1\rangle \leq \langle f_2, d_2\rangle$ iff $f_1 = f_2$ and $d_1 \sqsubseteq d_2$ (both for function symbols and nodes). As globals, we use a set V_2 which serves as descriptions of values of global variables and (classes of) heap cells:

Globals: $[x]$, x a global variable or (abstract) heap cell.

In interprocedural frameworks for sequential programs [25, 15, 23], the analysis can be specified by providing the following functions:

$$\begin{array}{lll} \mathbf{trans}_e & : \quad \mathbb{D}_1 \to \mathbb{D}_1 & (e \text{ a basic computation step}) \\ \mathbf{entry}_e & : \quad \mathbb{D}_1 \to \mathbb{D}_1 & (e \text{ a call edge}) \\ \mathbf{combine}_e & : \quad \mathbb{D}_1 \times \mathbb{D}_1 \to \mathbb{D}_1 & (e \text{ a call edge}) \end{array}$$

where the function $\mathbf{entry}_e$ abstractly passes the actual parameters to the called procedure and $\mathbf{combine}_e$ merges the effect of the called procedure into the state of the caller. For the concurrent framework which we need, this approach is not sufficient. Basic computation steps may depend on globals and als may have side effects to globals. Potential function calls may happen through pointers implying that we do not know the functions to be called in advance. Also, due to potential loss in precision, it may happen that we will not call a single function at a call edge — but a *set* of potentially called functions. Moreover, we have to take care also spawning of newly created threads.
Consider an edge $e = (u, \mathsf{lab}, v)$ where a basic computation step is performed, e.g., an assignment $x = exp;$. Then the abstract effect is described by a function $\mathbf{trans}_e$:

$$\mathbf{trans}_e \quad : \quad \mathbb{D}_1 \times (V_2 \to \mathbb{D}_2) \to \mathbb{D}_1 \times (V_2 \to \mathbb{D}_2)$$

The function $\mathbf{trans}_e$ takes the local and global state and returns the new local state together with the potential side-effects to the global variables. Given the function $\mathbf{trans}_e$, we obtain the following constraints:

$$\begin{array}{lll} \langle v, d\rangle & \Leftarrow & \lambda(\sigma,\tau).\ \mathbf{let} \ \ (b,\eta) = \mathbf{trans}_e\,(\sigma\langle u, d\rangle, \tau) \\ & & \qquad\quad\ \mathbf{in} \ \ \ (b, \emptyset, \eta) \\ & & \qquad\quad\ \mathbf{end} \end{array}$$

for every $d \in \mathbb{D}_1$.
Consider an edge $e = (u, \mathsf{lab}, v)$ where new threads are created. Then the abstract effect is determined relative to a function $\mathbf{create}_e$:

$$\textbf{create}_e \quad : \quad \mathbb{D}_1 \times (V_2 \to \mathbb{D}_2) \to \mathbb{D}_1 \times \mathbb{D}_1 \times 2^{\mathcal{F}} \times (V_2 \to \mathbb{D}_2)$$

The function $\texttt{create}_e$ takes the local and global state and essentially returns a new local state. Additionally, it provides a description of the actual parameters for the created threads and a set of functions potentially executed by threads created at this edge. Also, possible side-effects cannot be dispensed here, as thread creation in general may affect the global state, e.g., by storing the result value of the thread or the (abstract) id of the newly created thread. Given the function $\textbf{create}_e$, we obtain the constraints:

$$\begin{array}{lll} \langle v, d\rangle & \leftarrow & \lambda(\sigma,\tau).\ \begin{array}[t]{lll} \textbf{let} & (b, b_1, F, \eta) & = \textbf{create}_e\,(\sigma\langle u, d\rangle, \tau) \\ \textbf{and}\ s & & = \{\langle f, b_1\rangle \mid f \in F\} \\ \textbf{in} & (b, s, \eta) & \\ \textbf{end} & & \end{array} \end{array}$$

for all $d \in \mathbb{D}_1$.

Finally, we consider function calls. If u_0 is the initial program point of some procedure, the variable $\langle u_0, d\rangle$ should receive the value d:

$$\langle u_0, d\rangle \quad \leftarrow \quad \lambda(\_,\_).\ (d, \emptyset, \lambda\_.\bot)$$

Also, if v is a return point of the procedure p, then the value of $\langle v, d\rangle$ should contribute to the value of the abstract call $\langle p, d\rangle$. Thus,

$$\langle p, d\rangle \quad \leftarrow \quad \lambda(\sigma,\_).\ (\sigma\langle v, d\rangle, \emptyset, \lambda\_.\bot)$$

Now assume that the edge $e = (u, \mathsf{lab}, v)$ represents a function call. Similar to the sequential framework, the abstract effect is determined relative to the two functions $\textbf{entry}_e$ and $\textbf{combine}_e$. These, however, now have the types:

$$\begin{array}{lll} \textbf{entry}_e & : & \mathbb{D}_1 \times (V_2 \to \mathbb{D}_2) \to \mathbb{D}_1 \times 2^{\mathcal{F}} \\ \textbf{combine}_e & : & \mathbb{D}_1 \times \mathbb{D}_1 \to \mathbb{D}_1 \end{array}$$

Similar to the sequential framework, the function $\textbf{combine}_e$ specifies how the return values of called functions affect the local state. The function $\textbf{entry}_e$ now is more general. It not only returns for given states global environment the (abstract) description of the actual parameters, but also the set of functions which are potentially called at e. Given these functions, we obtain the following constraints: given by:

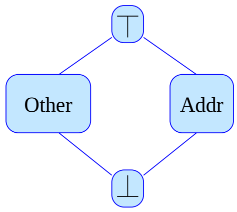
$$\begin{array}{lll} \langle v, d\rangle & \leftarrow & \lambda(\sigma,\tau).\ \begin{array}[t]{lll} \textbf{let} & (b_1, F) & = \textbf{entry}\,(\sigma\langle u, d\rangle, \tau) \\ \textbf{and}\ b & & = \bigsqcup\{\textbf{combine}\,(\sigma\langle u, d\rangle, \sigma\langle f, b_1\rangle) \mid f \in F\} \\ \textbf{in} & (b, \emptyset, \lambda\_.\bot) & \\ \textbf{end} & & \end{array} \end{array}$$

for all $d \in \mathbb{D}_1$.

Typically, the resulting constraint system $\mathcal{C}$ is un-tractably large (if not infinite). Consider, e.g., our introductory example from figure 3. Let us perform a simple data race analysis. For determining the mutexes which are locked and unlocked, we need a base constant propagation — at least for disambiguating the address arguments. Thus, our local state consists of three components:

– The first component states whether some thread has been created or not;
– The second component holds the set of definitely held mutex locks; whereas
– The third component records the (abstract) values of the local variables.

For our analysis, the domain $\mathbb{V}$ 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, it suffices to consider a lattice of the following structure:



where `Addr` and `Other` represent partial orderings for the description of addresses and other values, respectively. For a more complete analyzer as the one we have implemented, we need a more precise account of non-address `int`-values and also need sub-lattices for structures, unions and arrays, as well as for thread id's, mutexes, and condition variables. For addresses we choose `Addr` as the lattice of all subsets of some set `Base` of base addresses together with a special $\top$-element denoting the unknown address. In our minimalistic example, `Base` is given by:

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

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 $\subseteq$. As in our example there is always just one local variable, we can use $\mathbb{V}$ directly to describe the third component of local states.
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 function $\mathbf{create}_{(2,3)}$ for the edge $(2, 3)$ in our analysis is given by:

$$\mathbf{create}_{(2,3)}\,(\langle\_, M, v\rangle, \tau) = (\langle 1, M, \top\rangle, \langle 1, \emptyset, \&\mathsf{A}\rangle, \{\mathsf{inc}\}, \lambda\_.\bot)$$

The first component of the new local state is 1 — indicating that a thread has been created, the second component remains unchanged while the third one receives the value $\top$ here, since we do not know which thread id is created by create(). Furthermore, the possibly created thread always executes the function inc where the initial local state is given by the triple $\langle 1, \emptyset, \&\mathsf{A} \rangle$.

Discussion:

- Realistic data-race analyses as those we have implemented, should not only track sets of held locks but additionally also (abstract) thread id's. Accordingly, we do not record at each global the sets of definitely held mutex locks but a (partial) mapping of thread id's to corresponding sets of definitely held locks. Assuming that thread id's are unique, we flag a warning if the sets of locks corresponding to two different thread id's have an empty intersection.
- It must be emphasized that the outlined analysis is only correct under the assumption that mutex locking and unlocking never fails — which according to [4] is not necessarily true. Locking of mutexes, e.g., may fail due to external interrupts or because the lock already is held by the current thread. In an environment which guarantees absence of interrupts, it still makes sense to make the non-failure assumption — given that potential repetitions of locks are flagged as errors (which we do).
- For the case where interrupts cannot be excluded, we also 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 leave it then to the user which analysis better serves her needs.

Further tricks and details of our data-race analyzer, we will report elsewhere. Returning to our example program, we now aim at exploring the constraint variables from V_1 by starting from an initial call $\langle \mathsf{main}, a \rangle$, $a \in \mathbb{D}_1$ a suitable abstract value. In terminology of section 2, this means that we are interested in the least partial invariant (τ, V) of the constraint system with $\langle \mathsf{main}, a \rangle \in V$ – together with a (upto equivalence) least partial solution σ. This is precisely what our solver from the next section will compute.

4 Solving Side-Effecting Constraint Systems

Our algorithm[1] for computing partial invariants (together with corresponding partial solutions) for side-effecting constraint systems 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 before-hand 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`.

[1] In the sequel, we will use typewriter fonts to specify (fragments of) real code.

This basic scheme (for non-side-effecting constraints) must be enhanced as evaluation of the right-hand side `f` not only results in a contribution to the left-hand side `x`, but also in a set s of further variables which are to be explored and a map η of side effects. Technically, we represent the s by a list of variables and the map η by a list of pairs `(y,d)`. During constraint evaluation, we have to take care that solving is triggered for all variables in s. Also, after performing the side effects, all constraints possibly influenced by side-effected variables must be scheduled for re-evaluation. Typically, globals are accessed in many places, meaning that each of them may influence a huge number of fixpoint variables. Thus, immediate re-evaluation of invalidated constraints intuitively can result in "tight" looping with many repeated evaluations and small increments. Therefore, we propose a different iteration strategy here: we collect all constraints triggered for re-evaluation through side-effects. Only when the iteration through fixpoint variables has stabilized, we perform a next round on this collection.
In detail, the algorithm uses the following data structures:

```
type rhs        = (V1 -> D1) * (V2 -> D2)
                  -> D1 * V1 list * (V2 * D2) list
type constraint = V1 * rhs

val sigma1  :  V1 -> D1
val infl1   :  V1 -> constraint list
val todo    :  V1 -> rhs list

val sigma2  :  V2 -> D2
val infl2   :  V2 -> constraint list

val nextIteration : constraint list ref
```

The mapping `sigma1` and `sigma2` return the current values of fixpoint variables and globals. Initially, we assume that `sigma1 x` is undefined everywhere (i.e., equal to $\perp$), and `sigma2 z` equals the least element `D2.bot` of `D2`. In order to test whether `sigma1` is defined for some `x`, we use the function

```
val isNew : V1 -> boolean
```

The entries of `sigma1` for `x` can be set/updated to a value `d` by writing:

```
sigma1 x := d
```

Analogously, the values of `sigma2` can be modified. The mappings `infl1` and `infl2` represent the (dynamically tracked) variable dependences. More precisely, `infl1 y` returns the list of all constraints during whose evaluation the value of `y` has been accessed (analogously for `infl2 z`). Initially, all values `infl1 y` and `infl2 z` equal the empty list `[]`. Similarly to `sigma1`, we feel free to add further elements to lists `infl1 y` and `infl2 y` or to reset them to `[]`.
The mapping `todo` 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.
The data structure `nextIteration` collects constraints whose evaluation has been postponed. Initially, `nextIteration` is empty. The operation `add` adds a

list of constraint to this data-structure whereas the operation `get` returns its contents and resets it to the (reference to) the empty list:

```
fun add (nextIteration, constraints) = (
         nextIteration := constraints @ (!nextIteration))
fun get  nextIteration = (!nextIteration) before
                         (nextIteration := [])
```

The algorithm itself then consists of the following functions:

```
val solve   :  V1 -> unit
val merge   :  constraint * V1 list -> unit
val eval1   :  constraint -> V1 -> D1
val eval2   :  constraint -> V2 -> D2
val update  :  V2 * D2 -> unit
val loop    :  V1 list -> unit
```

The functions `update`, `eval1` and `solve` are mutually recursive. We start their explanation with the main function `solve` which is displayed in figure 4.

```
fun solve x = let
    val _ = if isNew x then
               sigma1 x := D1.bot
            else ()
    val work = (todo x before
                     (todo x := []))
    val old = sigma1 x
    val new = foldl (fn (f,old) => let
        val (d,spawn,side) = f (eval1 (x,f), eval2 (x,f))
        val _ = app update side
        val _ = app solve spawn
        in old D1.lub d
        end) old work
    in if new = old then ()
       else ( sigma1 x := new
            ; app solve (foldl merge []
                        (infl1 x before
                        (infl1 x := [])))
            )
    end
```

Fig. 4. The function `solve`.

The function `solve` when called with a variable `x`, first checks whether `x` has already a value. If not, `x` is initialized with some safe value, say `D1.bot`. Then all constraints for x which are currently scheduled for (re-) evaluation, are extracted from data-structure `todo`. Evaluating each of these constraints result in a contribution `d` to the new value of `x` together with a list `spawn` of new variables to

be considered as well a list `side` of side effects to globals. Individual side effects are dealt with through the auxiliary function `update` (see below), whereas (re-)evaluation is triggered for each variable in `spawn`. All contributions `d` to `x` are gathered into the new value `new`. The additionally necessary work depends on the relation between the two values `old` and `new`.
If the two values differ, we update the value `sigma1 x` to `new`. Moreover, we must re-evaluate all other constraints whose evaluation has accessed `x`. These constraints have been accumulated in the entry `infl1 x`. By extracting these constraints from `infl1 x`, we re-set this entry to the empty list `[]`. Now, folding the function `merge` over the extracted list of constraints, inserts these constraints into the data-structure `todo` while at the same time a (duplication-free) list of their left-hand sides is collected. Finally, we apply the function `solve` (tail-) recursively to all found left-hand sides.
The definitions of the functions `merge`, `eval1` and `eval2` are collected in figure 5. The function `merge` is used to merge new constraints into the `todo` data structure

```
...
and merge ((x,f), xs) = case todo x
              of []   => (todo x := [f] ; x::xs)
              |  fs   => (todo x := f::fs ;  xs)

and eval1 constraint y = ( solve y
                   ; infl1 y := constraint :: infl1 y
                   ; sigma1 y
                   )

and eval2 constraint y =
                   ( infl2 y := constraint :: infl2 y
                   ; sigma2 y
                   )
```

Fig. 5. The functions `merge`, `eval1` and `eval2`.

— while at the same time collecting the variables for which there is something to do. Thus, when applied to a constraint `(x,f)` and a list `xs` of already collected left-hand sides, the right-hand side `f` is added to the entry `todo x`. In order to avoid replication of variables, these are only collected, if the corresponding entry in `todo` has been empty before-hand. In this case, x::xs is returned, otherwise, just `xs`.
The functions `eval1` and `eval2` are used during evaluation of right-hand sides for wrapping accesses to entries of the mappings `sigma1` and `sigma2`. The wrapper `eval1` solves variables first before it returns their values. Furthermore, it records at every access of the variable `y`, the constraint for which `y` was queried. Thus, the application `eval1 constraint y` first calls `solve` for the variable `y`. Then it records the variable dependence by adding the constraint `constraint` to the set `infl1 y`. Finally, it returns the value of `sigma1 y`. The wrapper `eval2` does the

same thing for accesses to entries of the global environment `sigma2`. As we do not allow constraints for globals, there is no preceeding call to a solving routine here.
The definitions of the functions `update` and `loop` are collected in figure 6. The

```
...
and update (z,d) = let
    val old = sigma2 z
    val new = old D2.lub d
    in if old = new then ()
       else ( sigma2 z := new
            ; add (nextIteration, infl2 z)
            ; infl2 z := []
            )
    end

fun loop xs = let
    val _ = app solve xs
    val lhs = foldl merge [] (get nextIteration)
    in case lhs
       of []         => ()
       |  _          => loop lhs
    end
```

Fig. 6. The functions `update` and `loop`.

function `update` is used to process an individual update of a side-effected variable. Thus, when called with `(z,d)`, `update` adds `d` to the old value of `z`. If the value of `z` in `sigma2` has been changed, all constraints influenced by `z` must be re-evaluated. Therefore, these are added to the list `nextIteration`.
The main loop of the algorithm then is implemented through the function `loop`. When called with a list `xs` of variables, it first applies `solve` to all of them. As a result, the data-structure `nextIteration` may have received a list of constraints whose evaluation has been postponed while solving the `xs`. This list is merged into the mapping `todo` while at the same time the corresponding list `lhs` of variables occurring as left-hand sides is collected. If this list is empty, the fixpoint iteration is completed, and we exit. Otherwise, we (tail) recursively call `loop`.
We have the following theorem:

Theorem 2. *Assume that the lattices* $\mathbb{D}_1$, $\mathbb{D}_2$ *have finite height, and that we have started with an empty assignment* `sigma1`, *a least environment* `sigma2` *and a list* `xs` *enumerating the interesting variables from the set* X. *Then the following holds:*

1. *If the set* V_1 *of constraint variables is finite, the algorithm always terminates.*
2. *After termination,* `sigma2` *is a partial invariant, and the variable assignment* `sigma1` *is a corresponding partial solution of the constraint system.*

If the implementations of right-hand sides of constraints are "sufficiently well-behaving" w.r.t. their variable accesses and the constraint system is weakly monotonic, then we even find that the result is equivalent to the minimal partial solution (see [9] for an analogous result). We refrain from a detailed discussion here.

5 A Tiny Example

As an illustration, consider the C program from section 3 and assume that we apply the side-effecting solver from section 4 to the corresponding constraint system for potential data race detection. Initially, all fixpoint variables in V_1 and all globals have the value $\bot$. Then fixpoint iteration starts with variable $\langle \mathsf{main}, a\rangle$ where $a = \langle 0, \emptyset, \top\rangle$ meaning that initially, there is no other thread besides the main thread, that the set of held mutex locks is empty, and the local id of main has an unknown value. In order to determine the result value of the call, the solver successively descends through calls `solve` $\langle 4, a\rangle$, `solve` $\langle 3, a\rangle$, `solve` $\langle 2, a\rangle$ and `solve` $\langle 1, a\rangle$ to the call `solve` $\langle 0, a\rangle$. This descent is due to the function `eval1` which first evaluates a variable before it returns its value. 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 $\langle 1, a\rangle$ and $\langle 2, a\rangle$ where the latter returns the triple $(a, \emptyset, \{(\mathsf{z}, \langle \bot, \{\mathsf{null}\}\rangle)\})$. Thus, the variables $\langle 2, a\rangle$ and z receive the values a and $\langle \bot, \{\mathsf{null}\}\rangle$, 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 $\langle 3, a\rangle$. Here, we have to take the effect of the create edge into account. The corresponding constraint returns the triple $(a_1, \{\langle \mathsf{inc}, b\rangle\}, \emptyset)$ where $a_1 = \langle 1, \emptyset, \top\rangle$ represents the new value of the local of main, i.e., the value for $\langle 3, a\rangle$, and $b = \langle 1, \emptyset, \&\mathsf{A}\rangle$ is the new local state of the newly created thread.
Before ascending to the variable $\langle 4, a\rangle$, the solver triggers the evaluation of the variable $\langle \mathsf{inc}, b\rangle$. Thus, it descends into solving the variables $\langle 8, b\rangle$ down to $\langle 5, b\rangle$, respectively. As 5 is the entry point of the procedure inc, the variable $\langle 5, b\rangle$ receives the value &A (corresponding to the value of the formal me). Proceeding to the variable $\langle 6, b\rangle$, the solver executes the locking of the mutex at address &A. This results in the new local state $b_1 = \langle 1, \{\&\mathsf{A}\}, \&\mathsf{A}\rangle$. Now, the solver executes the constraint corresponding to the edge $(6, 7)$. This edge modifies the global z through the side effect $\{(\mathsf{z}, \langle \{\&\mathsf{A}\}, \top\rangle)\}$. Note that this is done by accessing the value of z which means that the current constraint has been recorded in the list `infl2 z`. The new abstract value for z is

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

Since the value of z has been changed, the complete list `infl2 z` is moved into the data-structure `nextIteration`. The local state for $\langle 7, b\rangle$ remains b_1. Finally, the solver ascends to the variable $\langle 8, b\rangle$ where the unlock is performed yielding the local state $b_2 = \langle 1, \emptyset, \&\mathsf{A}\rangle = b$. Finally, the solver ascends to the variable $\langle \mathsf{inc}, b\rangle$ for which it returns b.

Having finished this detour, the solver proceeds with the analysis of main by evaluating the constraint for $\langle 4, a\rangle$ 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 constraints which have been added to `nextIteration` are scheduled for re-evaluation. In our case, this is just the constraint corresponding to the edge $(6, 7)$. This constraint is added to the entry `todo` $\langle 7, b\rangle$. Calling the procedure `solve` for $\langle 7, b\rangle$ re-evaluates this constraint resulting in no new value for $\langle 7, b\rangle$ but again in the side-effect $\{(\mathsf{z}, \langle\{\&\mathsf{A}\}, \top\rangle)\}$. 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.

6 Experimental Evaluation

Based on the solver from section 4 and the interprocedural framework from section 3, we have implemented a program analyzer generator for C programs with pthreads. The implementation was done in Standard ML using the ckit [11] as our C frontend. For efficiency reasons we organized the analysis as a multi-stage procedure. In the first stage, we determine approximative data values for all globals. This first invariant then is used in the following stages which additionally track acquired mutex locks. In our practical experiments, we perform two variants of the latter analysis: while the first analysis, DR1, assumes that mutex locks always succeed, the second one, DR2, does not make this assumption.

We used the framework to implement various analyses for the detection of data races. 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 seven components ranging in size from 23,000 to 80,000 LOC (before pre-processing and excluding header files). The analysis was performed on an 1 GigaHertz Atlon with 1 GB memory using SUSE Linux and the SML compiler smlnj-110.0.7. Some characteristics of our experiments are reported in fig. 7. The first column of table 7 gives the names of the analyzed software components, the next three columns provide code statistics (lines of

bench	LOC	th	mtx	Base		DR1			DR2		
1	23704	4	14	.37	85190	.37	86313	8	.41	91022	9
2	31658	5	7	1.00	134821	.59	151052	28	1.22	186966	36
3	37047	1	7	1.17	205148	.20	82679	0	.25	93053	0
4	31870	4	8	.16	55078	.20	63677	7	.21	64136	7
5	29554	4	10	1.37	174743	2.56	291278	31	15.49	716220	38
6	45660	8	13	17.07	260644	1.43	223032	6	5.39	509274	20
7	79474	26	?	4.20	463437	?			?		

Fig. 7. Experimental results.

C code, numbers 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. 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 false alarms.

7 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 [19]. Quite a few analyses have been recently developed for optimizing synchronizations in Java [6, 3, 1, 20]. Others deal with a combined analysis of pointers and multi-threading [12, 26]. Jagannathan and Weeks analyze even higher-order functional languages but merge all calls to the same function into one [12].

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-sensitve pointer analysis for multi-threaded Ada [21]. It is also advocated by Naumovich et al. in [17]. Based on a *possibly runs in parallel* analysis [18], they construct an enriched control-flow graph for Java on top of which they model the dynamic thread behavior via *property automata*.

Obviously, the practicality of such an approach heavily relies on the "density" of the enriched control-flow graph. Only recently, Yahav has presented a framework for analyzing properties of multi-threaded Java programs based on three-valued logic [27]. 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.

Presently in our implementation, we assume that all global data potentially can be accessed and modified by all threads. Clearly, this often 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 [22]. It remains to see in how far this analysis problem can be formulated as an interprocedural invariant and thus be handled within our framework. If so, it clearly could be used to improve the precision of the computed results.

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 under 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. M. Alt and F. Martin. Generation of Efficient Interprocedural Analyzers with PAG. In *2nd Static Analysis Symposium (SAS)*, pages 33–50. LNCS 983, 1995.
3. 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.
4. D.R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.
5. P. Cousot. Invariance Proof Methods and Analysis Techniques for Parallel Programs. In A.W. Biermann, G. Guiho, and Y. Kodratroff, editors, *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Collier Macmillan Publishers, London, 1984.
6. P.C. Diniz and M.C. 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. N. Heintze, D. Oliva, and D. MacQueen. ckit 1.0, March 2000. Available from: `http://plan9.bell-labs-com/cm/what/smlnj/doc/ckit`.
12. S. Jagannathan and S. Weeks. Analyzing Stores and References in a Parallel Symbolic Languages. In *ACM Conf. on LISP and Functional Programming*, pages 294–305, 1994.
13. 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.
14. 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.
15. Jens Knoop and Bernhard Steffen. The Interprocedural Coincidence Theorem. In *4th Int. Conf. on Compiler Construction (CC)*, pages 125–140. LNCS 641, Springer, 1992.
16. Markus Mller-Olm and Helmut Seidl. On Optimal Slicing of Parallel Programs. In *ACM Symposium on Theory of Computing (STOC)*, pages 647–656, 2001.
17. 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.
18. 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.
19. M.C. Rinard. Analysis of Multithreaded Programs. In *7th Int. Static Analysis Symposium (SAS)*, pages 1–19. LNCS 2126, Springer Verlag, 2001.
20. 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.
21. R. Rugina and M.C. 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.
22. A. Salcianu and M.C. Rinard. Pointer and Escape Analysis for Multithreaded Programs. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, pages 12–23, 2001.
23. H. Seidl and C. Fecht. Interprocedural Analyses: A Comparison. *Journal of Logic Programming (JLP)*, 43(2):123–156, 2000.
24. H. Seidl and B. Steffen. Constraint-Based Inter-Procedural Analysis of Parallel Programs. *Nordic Journal of Computing (NJC)*, 7(4):375–400, 2000.
25. Micha Sharir and Amir Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.
26. 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.
27. 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.