Soundness of Data Flow Analyses for Weak Memory Models*

Jade Alglave, Daniel Kroening, John Lugton,
Vincent Nimal, and Michael Tautschnig

Department of Computer Science, University of Oxford, UK

Abstract. Modern multi-core microprocessors implement *weak memory consistency models*; programming for these architectures is a challenge. This paper solves a problem open for ten years, and originally posed by Rinard: we identify sufficient conditions for a data flow analysis to be sound w.r.t. weak memory models. We first identify a class of analyses that are sound, and provide a formal proof of soundness at the level of trace semantics. Then we discuss how analyses unsound with respect to weak memory models can be repaired *via* a fixed point iteration, and provide experimental data on the runtime overhead of this method.

1 Introduction

Modern computing systems frequently employ multiple CPU cores, generating strong demand for concurrent software that exploits multiple threads of execution for better performance. However, the concurrency model implemented by these architectures is a formidable challenge for the programmer: with a goal of improving throughput, modern multi-core or multiprocessor architectures such as Intel's x86 series or IBM's PowerPC relinquish the standard execution model known as *Sequential Consistency* (SC) [1], in favour of much weaker models [2, 3]. Multiprocessors featuring a *weak memory model* permit execution traces that do not correspond to any interleaving of the program's instructions, that is, the architecture does not implement SC.

Program bugs that relate to weak memory consistency are often difficult to reproduce and to diagnose. Fig. 1 shows a standard example to illustrate the problem. At line (a), processor P_0 writes the value 1 into memory address `x`; then at line (b) it reads from memory address `y` and writes the result into the processor-local register `r1`. Similarly, at line (c), processor P_1 writes the value 1 into memory address `y`; then at line (d) it reads from memory address `x` and writes the result into processor-local register `r2`. We underline the fact

Init: `x=0; y=0;`

P_0	P_1
(a) `x` $\leftarrow$ `1`	(c) `y` $\leftarrow$ `1`
(b) `r1` $\leftarrow$ `y`	(d) `r2` $\leftarrow$ `x`

Observed? `r1=0; r2=0;`

Fig. 1. Litmus test illustrating store buffering

* Supported by EPSRC under grants no. EP/G026254/1 and EP/H017585/1, by the ARTEMIS CESAR project, and under the European Union's Seventh Framework Programme (FP7/2007–2013)/ERC grant agreement no. 280053.

that registers are private to the processor holding them, *e.g.*, `r1` is private to P_0, whereas the memory addresses, *e.g.*, `x` and `y`, are shared. Assuming SC, at least one of the registers has to hold 1 after the execution of the four statements. However, when executing this program on a multi-core x86 or PowerPC machine, traces are observed in which *both registers hold* 0 in the final state [4]. This outcome can be caused by the store buffers implemented in these architectures. The situation is exacerbated by the fact that this non-SC observation only occurs in a small fraction of the executions. For instance, execution of the code of Fig. 1 using the `litmus` tool presented in [5] on an x86 system results in 99.13% SC-conforming traces among one billion executions.

These relaxations permitted in weak memory models affect the semantics of high-level languages such as Java [6] or C++ [7]. One way to address this issue is to restrict program analyses to programs that are guaranteed to only exhibit SC executions such as programs free of data races [8], or programs where memory barriers have been inserted to ensure that they only have SC executions [9, 10].

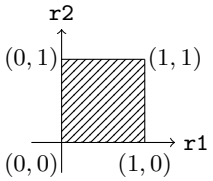
Yet we cannot restrict ourselves to this limited view on programs: engineers often choose to retain non-SC executions for performance reasons. *In other terms, we do* **not** *restrict our study to data-race free programs*. Consequently, effects relating to weak memory consistency need to be modelled appropriately in program analysis algorithms for concurrent software. The issue of soundness of program analyses w.r.t. weak memory models has been identified, among others, by Rinard, who wrote ten years ago in [11]:

> *"We suspect that many existing analyses are sound for programs with weak consistency models [...], but this soundness is clearly inadvertent, in some cases a consequence of the imprecision in the analysis, and not necessarily obvious to prove formally."*

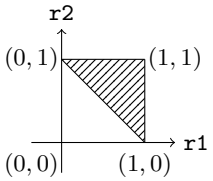
As an example, we first perform an *interval* analysis [12] on Fig. 1, to determine the possible values of `r1` and `r2`. We compute an interval for each variable. The *join* of two intervals yields the smallest interval that contains both of them. We consider all possible interleavings of statements of the two threads and compute the join over all these traces. For instance, for the traces $(a);(b);(c);(d)$ and $(c);(d);(a);(b)$ we obtain the intervals $[0,0]\times[1,1]$ and $[1,1]\times[0,0]$, respectively. The join $[1,1]\times[0,0]\sqcup[0,0]\times[1,1]$ yields the box $[0,1]\times[0,1]$, already including the result that can be derived from the other interleavings, *i.e.*, $[1,1]\times[1,1]$. More interestingly, this overapproximation also includes the additional value that one can observe on a weak memory model, *i.e.*, $(0,0)$.

As a second example of a program analysis, we consider the *octagon abstract domain* [13], which is a relational domain that describes (octagonal) faces of polyhedra. *Joining* two polyhedra in this abstraction consists in computing the smallest polyhedron which contains these polyhedra. For the two traces given above we obtain $\{1\}\times\{0\}\sqcup\{0\}\times\{1\} = \{\texttt{r1}+\texttt{r2}\leq 1,\ -\texttt{r1}-\texttt{r2}\leq -1,\ \texttt{r1}\leq 1,\ -\texttt{r1}\leq 0\}$, which concretizes to the diagonal line segment going from $(0,1)$ to $(1,0)$. *No join of interleavings, however, will include the point* $(0,0)$.

Fig. 2 provides a comparison of the results of intervals and octagons. In Fig. 2(c) we furthermore provide the code for reproducing these results using



(a) Box abstraction



(b) Octagon abstraction

```
var x,y:int; // shared memory
initial x==0 and y==0;
thread P0:
var r1:int; // P0 register
begin
  x = 1;
  r1 = y;
end
thread P1:
var r2:int; // P1 register
begin
  y = 1;
  r2 = x;
end
```

(c) Implementation of Fig. 1 for ConcurInterproc

Fig. 2. Running interval and octagon on Fig. 1 to compute the values of (r1, r2)

ConcurInterproc[1] [14]. The octagon domain is thus unsound w.r.t. weak memory models, whereas the (less precise) interval domain belongs to a class of analyses sound for weak memory models, as we show in this paper.

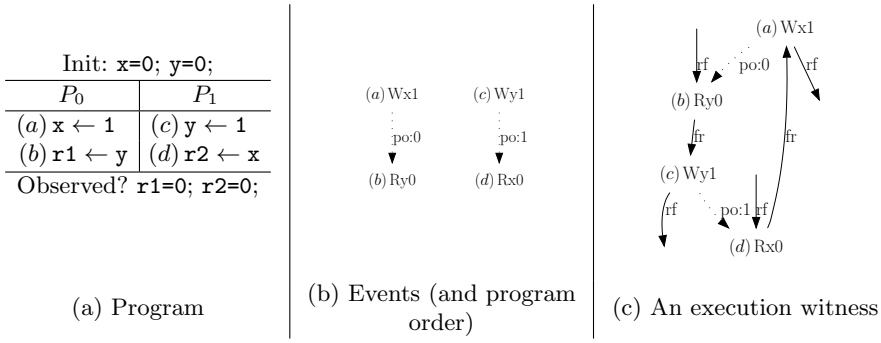
Few proofs of soundness of program analyses for weak memory models exist. In addition, existing proofs are usually tailored to a particular analysis and a particular memory model [15, 16]. These proofs thus offer only limited general insight into what makes an analysis sound for weak memory models.

Contributions We establish sufficient conditions for a data flow analysis to be sound w.r.t. weak memory models. We identify a large class of data flow analyses—the *non-relational* ones—that satisfy these conditions. These are guaranteed to be sound for a wide range of modern architectures, namely all those that respect the uniproc axiom as defined in [17, 4] and recalled in Sec. 2. Our results use trace semantics, hence are independent of the programming language and the specific representation of the concurrent program used in the analysis. Our classification confirms recent research results for specific analyses [15, 16, 18, 19] as part of a broader result. It also simplifies existing ad-hoc proofs, and provides proofs that new analyses are sound w.r.t. weak memory models.

We furthermore address the question of repairing an analysis that is unsound for weak memory models. We provide a general method to extend a sequentially sound *forward* analysis to an analysis for concurrent programs that is sound for weak memory models. We illustrate the method with the octagon domain.

We omit the proofs for brevity, but they can be found together with the details of our experiments at `http://www.cprover.org/wmm/`.

[1] `http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi`



(a) Program

(b) Events (and program order)

(c) An execution witness

Fig. 3. A program and a candidate execution

2 Background

To apply program analyses to concurrent programs running on modern multi-core processors or multiprocessors, we need to prove that these analyses are actually sound w.r.t. weak memory models. To describe weak memory models, we use the generic framework of Alglave *et al.* [4, 17], which covers a wide range of existing architectures, in particular x86-TSO [20] and a fragment of Power. We summarise the relevant parts of this framework.

2.1 Weak Memory Models

Events Instead of dealing directly with programs, we reason in terms of the *events* occurring in a program execution. An event e is a memory access, composed of a direction R (read) or W (write), an address $\mathrm{addr}(e)$, a value $\mathrm{val}(e)$, a processor $\mathrm{proc}(e)$, and a program location $\mathrm{loc}(e)$. We will use registers, which are processor-local (thread-local) variables, in place of values when the actual valuation is not known *a priori*. Note that an address always refers to shared memory, and thus never to a register. We represent each instruction by the events it issues. In Fig. 3, we associate the store (a) `x` $\leftarrow$ `1` on P_0 with the event $e = (a)\mathtt{Wx1}$. For this example we have $\mathrm{addr}(e) = \mathtt{x}$, $\mathrm{val}(e) = 1$, $\mathrm{proc}(e) = P_0$, and $\mathrm{loc}(e) = (a)$.We write $\mathbb{E}$ for the set of events. We write w (resp. r) for a write (resp. read), and e when the direction of the event is irrelevant.

Executions We associate a program with an *event structure* $E \triangleq (\mathbb{E}, \overset{\mathrm{po}}{\rightarrow})$, composed of its events $\mathbb{E}$ and the *program order* $\overset{\mathrm{po}}{\rightarrow}$, a per-processor total order over $\mathbb{E}$. In Fig. 3, the store (a) to `x` on P_0 is in program order with the read (b) from `y` on P_0, *i.e.*, $(a)\mathtt{Wx1} \overset{\mathrm{po}}{\rightarrow} (b)\mathtt{Ry0}$.

Given an event structure E, we represent an execution witness $X \triangleq (\overset{\mathrm{ws}}{\rightarrow}, \overset{\mathrm{rf}}{\rightarrow})$ of the corresponding program by two relations over $\mathbb{E}$: the *write serialisation* $\overset{\mathrm{ws}}{\rightarrow}$ is a per-address total order on writes, linking a write w to all other writes w' to the same address hitting the memory after w; the *read-from map* $\overset{\mathrm{rf}}{\rightarrow}$ links

a single write w to a read event r that reads from the address that w writes to. The relations $\overset{\text{ws}}{\rightarrow}$ and $\overset{\text{rf}}{\rightarrow}$ are the key objects for defining the validity of an execution, as explained below. We derive the *from-read map* $\overset{\text{fr}}{\rightarrow}$ from $\overset{\text{ws}}{\rightarrow}$ and $\overset{\text{rf}}{\rightarrow}$. A read r is in $\overset{\text{fr}}{\rightarrow}$ with a write w when r reads from the address of some write w' that hits the memory before w does: $r \overset{\text{fr}}{\rightarrow} w \triangleq \exists w'.\, w' \overset{\text{rf}}{\rightarrow} r \wedge w' \overset{\text{ws}}{\rightarrow} w$.

The observable result, `r1=r2=0`, shown in Fig. 3(a) corresponds to the execution of Fig. 3(c) if each address and register initially holds 0. If `r1=0` in the end, the read (b) obtained its value from the initial state, hence before the write (c) on P_1, thus $(b) \overset{\text{fr}}{\rightarrow} (c)$. Similarly, if `r2=0`, then $(d) \overset{\text{fr}}{\rightarrow} (a)$.

Uniprocessor Behaviour The condition $\text{uniproc}(E, X) \triangleq \text{acyclic}(\overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow} \cup \overset{\text{rf}}{\rightarrow} \cup \overset{\text{po-loc}}{\rightarrow})$ (where $\overset{\text{po-loc}}{\rightarrow}$ is the program order restricted to events with the same address) forces a processor in a multiprocessor context to respect the memory coherence widely assumed by modern architectures [21, 22, 2, 3].

This means that if a processor writes, *e.g.*, the value v to the memory location ℓ and then reads v' from ℓ, then the associated writes w and w' should be in this order in the write serialisation, *i.e.*, w' should not precede w. In Fig. 4, we have $(c) \overset{\text{ws}}{\rightarrow} (a)$ (by `x` final value) and $(a) \overset{\text{rf}}{\rightarrow} (b)$ (by `r1` final value). The cycle $(a) \overset{\text{rf}}{\rightarrow} (b) \overset{\text{po-loc}}{\rightarrow} (c) \overset{\text{ws}}{\rightarrow} (a)$ invalidates this execution: (b) cannot read from (a) as it is a future value of `x` in $\overset{\text{ws}}{\rightarrow}$. In every model of our framework, there is no valid execution which ends up with $x = 1$, `r1` $= 1$.

P_0	P_1
(a) `x` $\leftarrow$ `1`	(b) `r1` $\leftarrow$ `x`
	(c) `x` $\leftarrow$ `2`

Forbidden: `x=1; r1=1;`

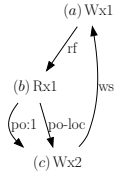


Fig. 4. An invalid execution, violating the uniproc condition.

The uniproc condition actually corresponds to checking that SC holds per address [17]. We rely heavily on this axiom in the proofs of this paper[2].

Architectures; validity of executions We define formally in [4, 17] the notions of *architecture* and *validity of an execution w.r.t. an architecture*, but we abstract them away in the present paper, for two reasons. First, the exposition of this paper does not need to detail them. Second, and more importantly, our results only require the architecture that we consider to respect the uniproc axiom.

Thus, in the following, we consider an abstract notion of *architecture*, which acts as a filter over executions. Given an architecture A, an event structure E and an execution witness X, we write $\text{valid}_A(E, X)$ when the execution (E, X) is valid on A. We only impose that $\text{valid}_A(E, X)$ implies $\text{uniproc}(E, X)$, *i.e.*, that a valid execution should pass the uniproc check.

We also abstract the notion of comparison over architectures of [4, 17]. Intuitively, an architecture A_1 is *weaker* than another one A_2 when the executions valid on A_2 are valid on A_1. Thus, SC is stronger than any other architecture.

[2] All the results presented here hold with a weaker version of uniproc, which allows us to embrace Sun's RMO in our framework. We omit this restriction for clarity and brevity, but more details can be found in [17, p.47–48].

2.2 Programs vs. Event Structures

Event structures describe programs in terms of their trace semantics. In the programs considered above, each right hand side of a store was a single concrete value, which immediately translated to a (concrete) event. To derive event structures from a description of general high-level programs, however, we proceed in two steps. Each control flow path at program level first translates to an *abstract event structure*, where events take the form of a direction and two variables. This allows us to translate, *e.g.*, a store $(a)\ x \leftarrow \sigma$ to the (abstract) event $(a)\mathrm{W}x\sigma$.

We write $\mathcal{E}$ for the set of all abstract event structures, $\mathbb{A}$ for the set of all addresses, and $\mathbb{V}$ for the set of all values. We define the type $\mathcal{R}$ of *results* (or valuations) as $\mathcal{R} \triangleq \wp(\mathbb{A} \times \mathbb{V})$, *i.e.*, a result is a set of pairs (x, v) where x is an address and v a value (we denote the powerset with $\wp(\cdot)$).

Given a specific language $\mathcal{L}$, we write $\mathbb{P}_{\mathcal{L}}$ for the set of all the programs which can be written in this language. We introduce $\alpha : \mathbb{P}_{\mathcal{L}} \to \wp(\mathcal{E} \times \mathcal{R})$, which maps a program $\mathcal{P}$ to corresponding abstract event structures and initial values, respecting the semantics of the language $\mathcal{L}$. Each event created by α is labelled by the program counter of the corresponding statement in $\mathcal{P}$.

Each abstract event structure induces multiple *concrete event structures* under a given set of initial valuations. That is, an abstract event $(a)\mathrm{W}x\sigma$ with $R = \{(\sigma, 0), (\sigma, 1)\}$ translates to concrete events $(a)\mathrm{W}x0$ and $(a)\mathrm{W}x1$. The set of all sets of concrete event structures is denoted by $\mathcal{E}_{\mathrm{conc}}$. We use the mapping $\mathrm{conc} : \mathcal{E} \to \mathcal{R} \to \mathcal{E}_{\mathrm{conc}}$ to translate abstract to concrete event structures. We require conc to yield a set of concrete event structures such that at least for each execution witness valid on an architecture A there is a concrete event structure.

We distinguish abstract from concrete event structures as follows: program analyses will be applied to abstract event structures, but reasoning about actual values will be performed in concrete event structures.

3 Soundness of Analyses on Weak Memory Models

We define an *analysis* $[\![\cdot]\!]$ as mapping abstract event structures and initial valuations to sets of pairs (i, r) where i is a program location (of type $\mathbb{L}$) and r is a result as defined in the preceding section. We make explicit the initial state of values of type $\mathcal{R}$, commonly being the empty set or the set of all possible values:

$$[\![\cdot]\!] : \mathcal{E} \to \mathcal{R} \to \wp(\mathbb{L} \times \mathcal{R})$$

Note that our definition captures *relational* analyses [23]; indeed the result type $\wp(\mathbb{L} \times \mathcal{R})$ can be rewritten as $\mathbb{L} \to \wp(\mathbb{A} \to \wp(\mathbb{V}))$.

Consider an abstract event $(a)\mathrm{W}x\sigma$ with initial valuations $R = \{(\sigma, 0), (\sigma, 1)\}$. We track the concrete values of x and the relation between x and σ as follows:

$$[\![(a)\mathrm{W}x\sigma, \{(\sigma, 0), (\sigma, 1)\}]\!] = \{((a), \{(x, 0), (\sigma, 0)\}), ((a), \{(x, 1), (\sigma, 1)\})\}$$

3.1 Definition of Soundness

Rinard and Rugina define in [24, A.3] an analysis to be *sound*

> *"[. . .] if it is at least as conservative as the result obtained by using the standard pointer analysis algorithm for sequential programs on all the interleavings of the legal executions."*

A *legal execution* corresponds to the execution of one thread. Thus their work assumes SC (*i.e.*, the interleaving semantics) as the execution model. We generalise their idea to weak memory models. Given an architecture A, we write $\text{values}_A(E, R)$ for the set of values that execution witnesses X can yield on A, where X is an execution witness associated to a concrete event structure $E' \in \text{conc}(E, R)$, *i.e.*, obtained from concretizing E with initial valuations R.

We write $<_X$ for the order on program locations induced by an execution X. We omit the formal definition of $<_X$ for brevity; it corresponds to $A\,.\,\text{ghb}(E, X)$ as defined in [17, 4]. Intuitively, it describes the order in which the memory events of X hit the memory. For example in Fig. 3(c), on an architecture $A \neq \text{SC}$—for otherwise the execution X depicted would not be valid—$<_X$ corresponds to $\{((d), (a))\} \cup \{((b), (c))\}$.

We write $\text{last}(r, i, x)$ when the location of x is less than (or unrelated to) (i) in $<_X$, and x is one of the last elements in the relation r, *i.e.*, there is no element x' such that $(x, x') \in r$. If a given write $w = (i)\text{W}xv$ is the last element in $\text{ws}(X)$ at location (i), then the value v is the *current value of* x *at location* (i). For example in Fig. 4, the current value of x at line (c) (resp. (a)) is 1 (resp. 2), for the last write to x at line (c) (resp. (a)) is the write $(c)\text{W}x2$ (resp. $(a)\text{W}x1$).

Thus, we define $\text{values}_A(E, R)$ as the set of possible results, *i.e.*, mappings of each address to its current value, at each location. In other words, the set $\text{values}_A(E, R)$ collects all the possible values in memory addresses that can arise in an execution (E, X) that is valid on A:

$$\begin{aligned}\text{values}_A(E, R) \triangleq \{(i, r) \mid \exists X.\, \text{valid}_A(\text{conc}(E, R), X) \wedge \forall x, v.\, (x, v) \in r \Rightarrow \\ \exists w.\, (\text{last}(\text{ws}(X), i, w) \wedge \text{addr}(w) = x \wedge \text{val}(w) = v)\}\end{aligned}$$

For example in Fig. 3, $\text{values}_{SC}(E, R)$ contains, for program location (a), the result $((a), \{(\texttt{r1}, 0), (\texttt{r2}, 0), (x, 1), (y, 0)\})$. Formally, we define soundness of over-approximating analyses for a weak architecture A as follows:

Definition 1. *An analysis* $[\![\cdot]\!]$ *is* A-sound *iff the result of* $[\![\cdot]\!]$ *on an abstract event structure* E *with initial values* R *describes a state space at least as large as that of* $\text{values}_A(E, R)$ *(with* $U \preceq V$ *iff* $\forall (i, r) \in U.\, \exists r'.(i, r') \in V \wedge r' \subseteq r$*):*

$$\text{sound}_A([\![\cdot]\!]) \;\triangleq\; \forall E, R.\, \text{values}_A(E, R) \preceq [\![E, R]\!]$$

We have, *e.g.*, $\{((a), \{(\texttt{r1}, 0), (\texttt{r2}, 0), (x, 1), (y, 0)\})\} \preceq \{((a), \{(\texttt{r1}, 0), (\texttt{r2}, 0)\})$. This means that we consider an analysis result to be A-sound if it is at least as conservative as taking all the values yielded by all valid executions on A.

Note that under-approximating analyses for SC are also under-approximating for all weak memory models, since for all weak architectures A, the values valid on SC are also valid on A, *i.e.*, $\text{values}_{\text{SC}}(E,R) \preceq \text{values}_A(E,R)$. We therefore focus the presentation on showing soundness of over-approximating analyses.

3.2 SC-soundness Entails *A*-soundness for Non-relational Analyses

We now define a particular class of program analyses by restricting the signature of the output of the analysis. We only consider analyses $\widehat{[\![\cdot]\!]}$ that map abstract event structures to pairs (i,r) where i is a program location and r a result, with the additional constraint that r *is a singleton*:

$$\widehat{[\![\cdot]\!]} : \mathcal{E} \to \mathcal{R} \to \wp(\mathbb{L} \times (\mathbb{A} \times \mathbb{V}))$$

This type can be rewritten as $\mathbb{L} \to (\mathbb{A} \to \wp(\mathbb{V}))$. In practice, we apply a *projection* to our general type of analyses to obtain *non-relational* ones [23]:

$$\text{projection}([\![\cdot]\!])(E,R) \triangleq \{(i,\{(x,v)\}) \mid \exists r.\,(x,v) \in r \wedge (i,r) \in [\![E,R]\!]\}$$

We restrict the type of values_A similarly by computing values_A and then using the projection abstraction given above. We write $\widehat{\text{values}_A}(E,R)$ to indicate this, *i.e.*, $\widehat{\text{values}_A}(E,R)$ is of type $\mathbb{L} \to (\mathbb{A} \to \wp(\mathbb{V}))$. In the example of Fig. 3, $\widehat{\text{values}_{\text{SC}}}(E,R)$ contains $\{((a),(\texttt{r1},0)),((a),(\texttt{r2},0)),((a),(x,1)),((a),(y,0))\}$.

We want to determine when a given analysis, although designed with SC in mind, is sound for a weak architecture A. For example, for the program given in Fig. 3, we have $\widehat{\text{values}_{\text{x86}}}(E,R) = \widehat{\text{values}_{\text{SC}}}(E,R)$ (with the initial state R mapping all variables to 0). Hence in this case, an analysis that computes at least $\widehat{\text{values}_{\text{SC}}}(E,R)$ is also sound for x86, since it also computes all the values that this specific program can yield on an x86 machine. We show that *any analysis* $\widehat{[\![\cdot]\!]}$ with (1) matching signature and (2) that is SC-sound as defined above satisfies this requirement. This means that collecting the values produced by the SC executions (*i.e.*, $\widehat{\text{values}_{\text{SC}}}(E,R)$) suffices to obtain the values yielded by a weaker model A. This property is guaranteed by the uniproc check as defined in Sec. 2, since uniproc means that SC holds per location. To prove this claim, we first show the inclusion of value sets:

Lemma 1. $\forall E,R.\, \widehat{\text{values}_A}(E,R) \subseteq \widehat{\text{values}_{SC}}(E,R)$

The lemma is sufficient to show our main theorem, which states that for a non-relational analysis $\widehat{[\![\cdot]\!]}$, its SC-soundness (*i.e.*, $\forall E,R.\, \widehat{\text{values}_{\text{SC}}}(E,R) \subseteq \widehat{[\![E,R]\!]}$) entails its A-soundness on any architecture A. That is to say, we show in Thm. 1 that a non-relational analysis, though defined with SC in mind, is sound on a weaker architecture A when this analysis collects at least all the values yielded by all the executions valid on SC. We formalise this as follows.

Theorem 1. $\forall \widehat{[\![\cdot]\!]} : \mathcal{E} \to \mathcal{R} \to \wp(\mathbb{L} \times (\mathbb{A} \to \mathbb{V}))$. $\mathrm{sound}_{SC}(\widehat{[\![\cdot]\!]}) \Rightarrow \mathrm{sound}_A(\widehat{[\![\cdot]\!]})$

The result is obtained by reasoning over traces, which is the most precise, yet not necessarily computable, representation for program executions. Hence our results are *independent of (1) programming language specifics* such as locks or dynamic synchronization primitives and hold for all other program representations, such as (concurrent) control flow graphs or Petri nets, for they are overapproximations of the sets of traces (*cf.* [25] for a discussion of representations for concurrent programs); *(2) analysis specifics* such as fixed point iteration strategies and sources of imprecision.

Note that for a relational analysis, its SC-soundness would not, in general, entail its A-soundness, for the weaknesses of multiprocessors' execution models is precisely observable *via* relations over variables, as shown in Fig. 3. We discuss means of obtaining an A-sound analysis from a relational analysis in Sec. 4.

Octagon and Box As seen in Sec. 1, the octagon abstract interpretation is not sound on weak memory models. Indeed, this analysis reasons over conjunctions of statements, *e.g.*, for Fig. 3, it computes values that `r1` and `r2` can have at the same time. More formally, in [13], Miné defines the octagon concretization function with $\mathcal{D}^+ : DBM \to \wp(\mathbb{A} \to \mathbb{V})$, where DBM is the set of difference-bound matrices m^+. There is one matrix m_i^+ per line i, and the concrete domain computation takes the form $\lambda i.\mathcal{D}^+(m_i^+) : \mathbb{L} \to \wp(\mathbb{A} \to \mathbb{V})$. We have $\wp(\mathbb{A} \times \mathbb{V}) \subsetneq \wp(\mathbb{A} \to \mathbb{V}) \subsetneq \wp(\wp(\mathbb{A} \times \mathbb{V}))$, hence octagon analyses cannot be represented with the non-relational analysis type, $\mathcal{E} \to \mathcal{R} \to \wp(\mathbb{L} \times (\mathbb{A} \to \mathbb{V}))$, but always with the relational analysis type, *i.e.*, $\mathcal{E} \to \mathcal{R} \to \wp(\mathbb{L} \times \wp(\mathbb{A} \times \mathbb{V}))$.

As we show in the introduction, the interval abstraction, however, collects along the way the values that `r1` and `r2` can have on a weak memory model. This is a non-relational analysis, expressible with $\mathcal{E} \to \mathcal{R} \to \mathbb{L} \to \wp(\mathbb{A} \times \mathbb{V})$, and we can deduce from Thm. 1 that it is sound for weaker memory models if originally implemented for SC.

3.3 Proving Soundness of Analyses over Programs

Thm. 1 gives sufficient conditions for an analysis over event structures to be A-sound. We explain here how this result transfers to programs.

Let $\mathcal{P}$ be a program written in a language $\mathcal{L}$. To express the soundness of a program analysis $[\![\cdot]\!]_{\mathcal{L}}$, we require $\mathrm{values}_A(E, R)$ and $[\![\mathcal{P}]\!]_{\mathcal{L}}$ to have the same type $\wp(\mathbb{L} \times \mathcal{R})$, with $\mathbb{L}$ being the program counters of statements in program $\mathcal{P}$. As above, we define $\widetilde{\mathrm{values}}_A(\mathcal{P})$ as the values yielded by executions of $\mathcal{P}$ on A, *i.e.*, $\widetilde{\mathrm{values}}_A(\mathcal{P}) \triangleq \bigcup_{(E,R)\in\alpha(\mathcal{P})} \mathrm{values}_A(E, R)$ (recall that $\alpha : \mathbb{P}_{\mathcal{L}} \to \wp(\mathcal{E} \times \mathcal{R})$ maps a program $\mathcal{P}$ to corresponding abstract event structures and initial values, w.r.t. the semantics of the language $\mathcal{L}$).

Hence the A-soundness of a program analysis is merely a lifting of the A-soundness of the corresponding event structure analysis:

$$\widetilde{\mathrm{sound}}_A([\![\cdot]\!]_{\mathcal{L}}) \triangleq \forall \mathcal{P}.\, \widetilde{\mathrm{values}}_A(\mathcal{P}) \preceq [\![\mathcal{P}]\!]_{\mathcal{L}}$$

Therefore, the A-soundness of SC-sound non-relational program analyses holds as a corollary of Thm. 1:

Corollary 1. $\forall [\![\cdot]\!]_{\mathcal{L}} : \mathbb{P}_{\mathcal{L}} \to \wp(\mathbb{L} \times (\mathbb{A} \times \mathbb{V})).\ \widetilde{\text{sound}}_{SC}([\![\cdot]\!]_{\mathcal{L}}) \Rightarrow \widetilde{\text{sound}}_{A}([\![\cdot]\!]_{\mathcal{L}})$

Rugina and Rinard's Pointer Analysis Rugina and Rinard define in [24] a non-relational analysis (denoted RR in the following) for a subset of C with basic pointer assignments and control flow instructions. They prove that $RR(\mathcal{P})$ contains all the values appearing in the interleavings of the program $\mathcal{P}$, *i.e.*, RR is SC-sound. Thus by Cor. 1, RR is sound for memory models weaker than SC.

4 Repairing Unsound Analyses

We have shown that a non-relational analysis is A-sound if it is SC-sound. Yet some analyses, such as the octagon one, cannot be defined in the non-relational framework. Using the projection abstraction defined in Sec. 3.2, we may turn a relational analysis into a non-relational one. Thus, projecting an unsound analysis makes it sound (provided it is SC-sound) by Thm. 1.

Yet, this projection is very coarse, as it breaks all the relations over variables maintained by the analysis (*cf.* [13] for an example of the projection from octagon to interval). We present in the following a method to ensure the soundness of an analysis w.r.t. weak memory models that preserves the relational type of the analysis and conserves some of its precision. This method, which we call *repair loop*, is already implemented in several existing analyses, for performance reasons. Its consists of analysing each thread separately to capture the values the memory locations get, then feeding back the collected possible values to each of the other threads to simulate the effects of thread interference (*cf.* Sec. 6 for a discussion of several approaches already following this idea). We choose to simulate the process by:

1. building enough *concatenations of the threads* of a program;
2. analysing each of these as a sole thread *without killing any values.*

Concatenations of Threads We now assume that our event structures are *finite*, *i.e.*, have an arbitrary large yet finite number of events. We say that an event structure is a *thread* when all of its events belong to the same processor. Given an event structure E, a thread corresponds to the restriction of E to the events that run on processor p, written E_p. The *sequence* of two threads E_i and E_j is itself a thread, in the sense that it has only one processor; it gathers both the events of E_i and E_j, and its program order corresponds to the program order of E_i followed by the program order of E_j. We write e_i (resp. e_j) for the last (resp. first) event in program order on E_i (resp. E_j). We write $\text{evts}(E)$ (resp. $\text{procs}(E)$) for the events (resp. processors) of a given event structure E:

$$E_i ; E_j \triangleq \big(\text{evts}(E_i) \cup \text{evts}(E_j), \text{po}(E_i) \cup \text{po}(E_j) \cup \{(e_i, e_j)\}\big)$$

We further define the *concatenation* of n threads of an event structure E as:

$$\text{concat}(E, n) \triangleq \{E_{\text{big}} \mid \exists T_1, \ldots, T_n.\, E_{\text{big}} = T_1; \ldots; T_n \wedge \\ \forall i.\, \exists p \in \text{procs}(E).\, T_i = E_p\}$$

We prove that for a finite E there exists an integer n_{SC} (bounded by the cardinality of $\text{evts}(E)$) such that *each interleaving (an SC-valid execution) can be found as a subsequence* of some E_{big} in $\text{concat}(E, n_{\text{SC}})$. For example in Fig. 3, we simulate the interleaving $(a), (b), (c), (d)$ by building the concatenation $P_0; P_0; P_1; P_1$. Note that, since the E_{big} are themselves threads, we can analyse them with a sequential analysis. Thus, analysing all the E_{big} of $\text{concat}(E, n_{\text{SC}})$ gives us all the possible values yielded by the interleavings of E.

Analyse without Killing We define here what it means to analyse one thread without killing any values. We give in Alg. 1 the code of the recursive function awk. The function applies the analysis on the thread and propagates its relations to collect all the results.

$$\begin{array}{l}\text{awk } (\llbracket\cdot\rrbracket)\ (E,V) \triangleq \textbf{ if } \text{evts}(E) = \emptyset \textbf{ then } \emptyset \textbf{ else} \\ \quad \textbf{let } e = \text{first}(\text{evts}(E)) \textbf{ in} \\ \quad\quad \textbf{let } S = \llbracket e, V\rrbracket \textbf{ in} \\ \quad\quad\quad \textbf{let } S_R = \{R \mid \exists i.\, (i, R) \in S\} \textbf{ in} \\ \quad\quad\quad\quad S \cup \bigcup_{R \in S_R} \text{awk } (\llbracket\cdot\rrbracket)\ (\text{succ}(E, e),\ (V \cup R))\end{array}$$

Algorithm 1: awk function

This analysis is performed one event at a time (note that an event can change at most one value in memory). To keep the values which might get killed by a new event, at any given iteration, we do not only propagate the resulting relation R, but the union of this new result with the previous one V. This means that a future event will have access to the values of some previous result V computed during the analysis of the thread.

For example, the non-SC result $\{(\texttt{r1}, 0), (\texttt{r2}, 0)\}$ of Fig. 1 is not generated by octagon (written Oct in the following). Indeed, if we perform Oct on P_0, starting with $V = \{(x, 0), (y, 0), (\texttt{r1}, 0), (\texttt{r2}, 0)\}$, the only value that x can hold at line (b) is 1. Yet, the non-SC result is covered by awk(Oct) applied to the interleaving $(a), (b), (c), (d)$ of the code of Fig. 1. Indeed, if we apply awk(Oct) with the same initial result V, we get $R = \{\{(x, 1), (y, 0), (\texttt{r1}, 0), (\texttt{r2}, 0)\}\}$ as the result of $\text{Oct}((a), V)$ for the first event. Then, we compute $V' = R \cup V$ and propagate it to line (b), *i.e.*, we compute $\text{Oct}((b), V')$. This means that the event at line (b) has access to the value $(x, 0)$, for it appears in V'.

Fig. 5 gives a call graph of awk(Oct), on the interleaving $(a), (b), (c), (d)$ of the code of Fig. 1. A sequence $v_x v_y v_1 v_2$ represents $(x, y, \texttt{r1}, \texttt{r2}) = (v_x, v_y, v_1, v_2)$. The non-framed sequences are the ones that we propagate at every call of awk(Oct), and the framed ones are the results that we return for every event.

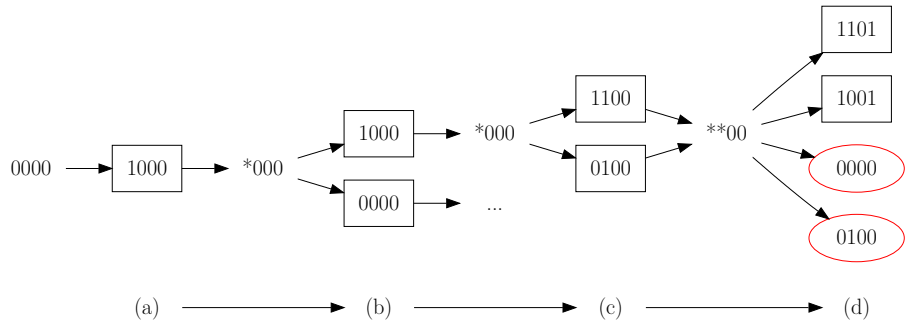


Fig. 5. Call graph of awk (Oct) ((*a*); (*b*); (*c*); (*d*), 0000).

Observe that we obtain the results 0000 and 0100 (in the ellipses) for event (d), both of them representing $(\texttt{r1}, \texttt{r2}) = (0, 0)$, *i.e.*, the non-SC behaviour. Indeed, when awk (Oct) $((d) : \texttt{r2} \leftarrow x, V)$ is executed, we not only have access to the previously computed result—as we would if we were directly applying Oct to the program—but rather to the union of all the previous results, including the initial one, where x holds 0.

Repair Loop Finally, we define the repair loop as a transformation of an analysis of type $\mathcal{A}$ to a new analysis. The repair loop builds n_{SC} concatenations of the threads of E to simulate all the interleavings of E, then analyses them without killing any values, and finally takes the union of the results:

$$\text{repair-loop}([\![\cdot]\!])(E, V) \triangleq \bigcup_{E_{\text{big}} \in \text{concat}(E, n_{\text{SC}})} \text{awk}([\![\cdot]\!])(E_{\text{big}}, V)$$

Revisiting the above example, applying repair-loop(Oct) to the code of Fig. 1 will simulate all the interleavings of the program. In particular, as we explained two paragraphs above, it simulates the interleaving $(a), (b), (c), (d)$ by the concatenation $P_0; P_0; P_1; P_1$. Then, it runs awk(Oct) on this concatenation, and since awk does not kill any value, we obtain the non-SC result $(\texttt{r1}, \texttt{r2}) = (0, 0)$, as explained in the preceding paragraph, and shown in Fig. 5.

5 Specification vs. Implementations of the Repair Loop

We gave in Sec. 4 a *specification* of the repair loop. Actual implementations are likely to use more scalable techniques such as fixed point computations. We leave the proof that such an implementation matches the specification of Sec. 4 for future work. Yet, we performed experiments to see how much an implementation of the repair loop would impair the performance of an analysis.

Experimental cost of the repair loop We showed that the points-to analysis of Rugina and Rinard is sound w.r.t. weak memory models even if performed with trace semantics, where the most precise results would be obtained. Yet, the analysis is implemented using a fixed point computation over the concurrent control flow graph instead of reasoning over all possible traces. As we showed above, the imprecision incurred by this fixed point iteration can be used to repair otherwise unsound analyses. A crucial question is, however, how many iterations are needed to arrive at a fixed point. We have shown that there exists a finite upper bound. It remains to be seen whether fewer iterations suffice in practice.

A similar study of the cost of such a fixed point iteration has recently been undertaken by Miné [16], and our experiments confirm his results: we use three sets of benchmarks that were previously used as case studies on the analysis of concurrent software: (1) concurrency bug patterns from the Apache web server as used in [26] (atom001, atom001a, atom002, atom002a, banking/av, banking/no_av, banking/some_av), (2) the banking and indexer examples from [27] (banking and indexer), and (3) several Linux device drivers together with nondeterministic environments as generated by DDVerify [28]. The detailed results and the source code of all experiments together with our implementation of Rugina and Rinard's points-to analysis is available at `http://www.cprover.org/wmm/`.

For several benchmarks of different origin our results confirm the findings of Miné: the typical number of iterations to reach a fixed point is very low, in fact it is always 2 in our samples. To study the overhead of repair iterations we summed up the time spent in all but the first iteration for each thread. We observe that this time overhead is very small with at most 0.034 seconds.

6 Related Work and Conclusion

We refer the reader to [29, 30] for an overview of the issues related to weak memory models. Program analyses for concurrent programs running on weak architectures have recently been considered by Ferrara [15] and Miné [16].

To the best of our knowledge, however, there is no general result on the soundness of program analyses for weak memory models. Both Ferrara and Miné describe extensions of the abstract interpretation framework to concurrent programs. In contrast to our work, which is generic, [15] explicitly focuses on an over-approximation of the Java memory model. Soundness w.r.t. the memory model is achieved by a fixed point iteration that implements the repair loop described in our paper. Miné [16] describes an extension of abstract interpretation to programs with a fixed number of threads and shared memory. He uses a fixed point iteration that is similar to the approach described by Rugina and Rinard [24] to compute a safe over-approximation of all possible interleavings. Furthermore Miné proves these results to be sound for a class of weak memory models specified as program transformations. Unlike our work, which is based on a framework for weak memory models that provably embraces several existing models, the modelling power of these transformations is unclear. With the re-

Analysis	Soundness w.r.t. weak memory models
Knoop *et al.* [33]	yes (separable)
Chugh *et al.* [34]	yes (if no datarace)
Steensgaard [35]	yes (flow-insensitive)
Miné [16]	yes
Rugina and Rinard [24]	yes
Jeannet [14]	no
Ferrara [15]	yes on Java Memory Model
Farzan and Kincaid [25]	yes (separable)
Khedker and Dhamdhere [36]	separable: yes; non-separable: not in general
Constant propagation [32]	yes (non-relational)

Fig. 6. Soundness of some concurrent analyses w.r.t. weak memory models

sults presented in the present paper it follows immediately that Miné's analysis extends from sequential consistency to weak memory models by the repair loop.

Sevcik and Vafeiadis *et al.* [18, 19] prove the correctness of a compiler for concurrent C programs towards x86 assembly, targeting the TSO model of [31]. Thus, they have to prove that analyses such as *constant propagation* [32] preserve the semantics from the source to the target program, which requires proving properties similar to ours. Since constant propagation is non-relational, we not only showed its soundness for TSO, but also for a large class of other models.

Several other analyses have been extended from sequential programs to the concurrent setting without explicitly discussing the effect of weak memory models. In the following, we survey their soundness w.r.t. weak memory models in the light of the results presented here. We summarise this discussion in Fig. 6.

We already discussed Rugina and Rinard's analysis [24] in Sec. 3.

Jeannet [14] presents the stack abstraction underlying ConcurInterproc, which can be combined with data abstraction domains such as octagons [13] or convex polyhedra [37] to apply abstract interpretation to parallel programs with a fixed number of threads. This approach is not generally sound for weak memory models as shown in the introduction.

Khedker and Dhamdhere [36] give a definition of separability for data flow analyses, *i.e.*, analyses where each data flow fact may be tracked in isolation, independently of the valuations of other data flow facts. Although separability is a concept independent of an analysis being (non-)relational, all separable problems can be expressed with the type that we proved to be sound in Thm. 1. For non-separable problems, we are unable to make such a general statement.

Most notably, bit-vector analyses are separable data flow analysis problems. Therefore the approaches described by Knoop *et al.* [33] and Farzan and Kincaid [25], who present methods of adapting a unidirectional bit-vector analysis designed with sequential programs in mind for use with multi-threaded programs, are immediately sound for weak memory models if sound for SC.

As another well-established classification of analyses consider flow sensitive vs. flow insensitive analyses. Rinard observes in [11] that flow insensitive analyses

such as Steensgaard's pointer analysis [35] are SC-sound. Hence by Thm. 1, we conclude that they are also sound for weak memory models.

The provable soundness of both bit-vector analyses and flow insensitive analyses is of uttermost practical importance as analyses of these kinds are used in optimizing compilers. Although today's compilers do not yet implement optimizations for multi-threaded programs in a concurrency-aware fashion, our results show that it would be safe to add such extensions.

Future Work While we already have a strong result for non-relational analyses, we would like to further refine our results for relational ones, *e.g.*, by lifting the restriction on the analysis being forward. Moreover, we intend to exercise our specification of the repair loop as given in Sec. 4, by proving that existing implementations, *e.g.*, Rugina and Rinard's, actually satisfy this property.

Acknowledgements We thank Vijay D'Silva, Peter Sewell, Viktor Vafeiadis and Thomas Wahl for invaluable discussions and comments.

References

1. Lamport, L.: How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. IEEE Trans. Comput. **46**(7) (1979) 779–782
2. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual, Vol. 3A, rev. 30. (March 2009) `intel.com/products/processor/manuals`.
3. IBM: Power ISA Version 2.06B. (July 2010) `power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf`.
4. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in Weak Memory Models. In: CAV, Springer (2010)
5. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Litmus: Running Tests Against Hardware. In: TACAS, Springer (2011)
6. Manson, J., Pugh, W., Adve, S.V.: The Java Memory Model. In: POPL. (2005)
7. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: PLDI. (2008)
8. Adve, S.V., Hill, M.D.: Weak ordering – A new definition. In: ISCA. (1990)
9. Burckhardt, S., Alur, R., Martin, M.K.: Checkfence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI. (2007)
10. Alglave, J., Maranget, L.: Stability in Weak Memory Models. In: CAV, Springer (2011)
11. Rinard, M.C.: Analysis of multithreaded programs. In: SAS, Springer (2001)
12. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: International Symposium on Programming, Dunod (1976)
13. Miné, A.: The octagon abstract domain. In: Workshop on Analysis, Slicing, and Transformation (AST), IEEE (2001)
14. Jeannet, B.: Relational interprocedural verification of concurrent programs. In: SEFM, IEEE (2009)
15. Ferrara, P.: Static analysis via abstract interpretation of the happens-before memory model. In: TAP, Springer (2008)
16. Miné, A.: Static analysis of run-time errors in embedded critical parallel C programs. In: ESOP, Springer (2011)

17. Alglave, J.: A Shared Memory Poetics. PhD thesis, Université Paris 7 and INRIA (2010) http://moscova.inria.fr/~alglave/these.
18. Sevcik, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. In: POPL. (2011)
19. Vafeiadis, V., Zappa Nardelli, F.: Verifying fence elimination optimisations. In: SAS. (2011)
20. Sewell, P., Sarkar, S., Owens, S., Zappa Nardelli, F., Myreen, M.: x86-TSO: a Rigorous and Usable Programmer's Model for x86 Multiprocessors. In: CACM. (2010)
21. SPARC: SPARC Architecture Manual Versions 8 and 9. (1992 and 1994) sparc.org/standards/V8.pdf and sparc.org/standards/SPARCV9.pdf.
22. Compaq: Alpha Architecture Reference Manual, Fourth Edition. (2002) download.majix.org/dec/alpha_arch_ref.pdf.
23. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
24. Rugina, R., Rinard, M.C.: Pointer analysis for multithreaded programs. In: PLDI. (1999)
25. Farzan, A., Kincaid, Z.: Compositional bitvector analysis for concurrent programs with nested locks. In: SAS, Springer (2010)
26. Wang, C., Limaye, R., Ganai, M.K., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: TACAS, Springer (2010)
27. Wang, C., Kundu, S., Ganai, M.K., Gupta, A.: Symbolic predictive analysis for concurrent programs. In: FM, Springer (2009)
28. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent Linux device drivers. In: ASE, ACM (2007)
29. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. IEEE Computer **29** (1995) 66–76
30. Adve, S., Boehm, H.J.: Memory Models: A Case for Rethinking Parallel Languages and Hardware. To appear in CACM.
31. Owens, S., Sarkar, S., Sewell, P.: A Better x86 Memory Model: x86-TSO. In: TPHOL. (2009)
32. Callahan, D., Cooper, K.D., Kennedy, K., Torczon, L.: Interprocedural constant propagation. In: SIGPLAN Symposium on Compiler Construction. (1986)
33. Knoop, J., Steffen, B., Vollmer, J.: Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. ACM Trans. Program. Lang. Syst. **18**(3) (1996) 268–299
34. Chugh, R., Voung, J.W., Jhala, R., Lerner, S.: Dataflow analysis for concurrent programs using datarace detection. In: Programming Language Design and Implementation (PLDI), ACM (2008) 316–326
35. Steensgaard, B.: Points-to analysis in almost linear time. In: POPL. (1996)
36. Khedker, U.P., Dhamdhere, D.M.: A generalized theory of bit vector data flow analysis. ACM Trans. Program. Lang. Syst. **16**(5) (1994) 1472–1511
37. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL. (1978)