2008 Sixth IEEE International Conference on Software Engineering and Formal Methods

Static analysis of the determinism of multithreaded programs

Pietro Ferrara *
École Polytechnique
F-91128 Palaiseau (France)
Dipartimento di Informatica
Università Ca' Foscari di Venezia
I-30170 Venezia (Italy)
Pietro.Ferrara@polytechnique.edu

Abstract

Threads communicate implicitly through shared memory. Because of the random interleaving during their parallel execution, nondeterministic behaviors possibly arise that is why multithreaded programming is strictly more difficult than programming in sequential languages. Moreover the random interleaving may lead to subtle bugs, that are really hard to be detected and fixed. We propose a novel deterministic property focused on multithreading. We define it as difference among concrete traces, and then we abstract it in two separate steps in order to statically analyze it. At the intermediate level of abstraction, we propose the new idea of weak determinism. We sketch how the proposed property may be used in order to semi-automatically parallelize sequential programs. Finally, we present some experimental results when applying the analysis to a set of well-known benchmarks. We believe that our approach, dealing directly with the source of the problem (i.e. the nondeterministic interactions via shared memory) is in position to bypass the actual limits of the static analysis of multithreaded programs, mostly focused on properties like data race condition and absence of deadlocks.

1. Introduction

While the improvement of single-core processors performances is slowing down, multi-core solutions appear to be the most promising way to extend the Moore's Law into the future [10]. The current trend of the CPU market underlines this idea: for instance the new family of Intel® processors (Intel® Core™) is dual or quad core, and the same happens for the AMD™ ones. Even if any application that runs on a single-core processor runs also on a multi-core one, in order to fully take advantage of these capabilities the applications must be optimized for multithreading. It is necessary to develop programs with explicit parallelism, otherwise a single application would not take relevant advantage from the multi-core hardware.

For developers, reasoning about concurrency and multithreading is strictly more difficult than working on sequential programs [18]. A first level of complexity is due to the interleaving of the executions of different threads, and to their implicit communications through shared memory. A more subtle level of complexity is induced by the specification of the memory model of the programming language [13], that is often not completely clear and understandable for the developer.

Both these issues are related to a common problem: the non-determinism induced by the parallel execution of many threads. In fact, some unexpected and counter-intuitive behaviors arise, like the ones due to the local optimization of compilers, invisible at single thread level.

A first way to restrict the non-determinism of multithreaded programs has been the static or dynamic checking on the concurrent actions [16]. Another way has been to define specific models that are in position to guarantee more restrictive multithreaded executions. Even if the work of the researchers in this field has been huge, there are still important limits in the proposed static analyses [18].

In this context, the contribution of this paper is mainly addressed into three distinct directions: (i) defining and abstracting in two steps the deterministic property in order to statically analyze it; (ii) defining on the first level of approximation the novel idea of weak determinism; (iii) sketching how determinism and its relaxations may be used in order to semi-automatically parallelize sequential programs.

In this way, we first define the determinism on a concrete domain and semantics, and then we abstract it, proving for-

*Work partially supported by MIUR PRIN'07 Project SOFT - Tecniche formali orientate alla sicurezza

978-0-7695-3437-4/08 $25.00 © 2008 IEEE
DOI 10.1109/SEFM.2008.14

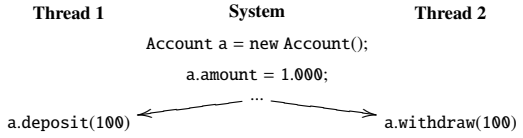
mally the soundness of our approach. In particular, at the first level of abstraction for each variable we trace a value for each thread that may have written it, while at the second level we trace a single value that approximates all the possible values and the set of threads that may have written in parallel on that variable.
Then we define the weak determinism, a novel idea that can be seen as a way of relaxing the deterministic property. We sketch how the deterministic property may be used in order to semi-automatically parallelize sequential programs. The analysis of these properties has been implemented, and we report some experimental results when applying it to a set of well-known benchmarks. As far as we know, our work is the first attempt to define and statically analyze the determinism of a multithreaded program generic on the programming language.

The rest of this section introduces a running example, some basic concepts of abstract interpretation, and the notation. Section 2 presents the concrete semantics, while Section 3 and 4 present respectively the first and second level of abstraction. The deterministic property is defined by Section 5, while the weak determinism is introduced in Section 6. Section 7 sketches how to use the deterministic property in order to semi-automatically parallelize sequential programs. Section 8 presents the experimental results. The related works are presented by section 9, and Section 10 concludes.

1.1. The running example

```
public class Account {
 public int amount=0;
 public void withdraw(int money) {
  synchronized(this) {
   account.amount-=money;
  }
 }
 public void deposit(int money) {
  synchronized(this) {
   account.amount+=money;
  }
 }
}
```

(a) The class Account



(b) Multithreaded execution

Figure 1. A running example

In order to illustrate the concepts presented throughout all the paper, we will always refer to the following running example written in Java style code.
Figure 1.(a) depicts the class `Account` that stores in a protected field `amount` an amount of money. Two methods are defined (`withdraw` and `deposit`), and they are used to withdraw and deposit money; they are both synchronized on the monitor defined on the current object. Figure 1.(b) depicts the multithreaded execution: the system is going to execute in parallel a deposit of 100€ and a withdraw of the same amount of money. At the beginning the amount of money contained in the account is 1.000€.

1.2. Abstract interpretation

Abstract interpretation is a theory to define and soundly approximate the semantics of a program [3, 4]. Roughly, a concrete semantics, aimed at specifying the runtime properties of interests, is defined; then it is approximated through one or more steps in order to finally obtain an abstract semantics that is computable, but still precise enough to capture the property of interest. The abstract semantics must be composed of an abstract domain, and an abstract transition function. Usually, each state of the concrete domain is composed by a set of elements (e.g. all the possible computational states), that is approximated by an unique element in the abstract domain.
Formally, the concrete domain D forms a complete lattice $\langle \mathsf{D}, \sqsubseteq_\mathsf{C}, \bot_\mathsf{C}, \top_\mathsf{C}, \sqcup_\mathsf{C}, \sqcap_\mathsf{C} \rangle$. On this domain, a semantics $\mathbb{S}$ is defined. The concrete elements are related to the abstract domain $\langle \mathsf{D}^\#, \sqsubseteq_\mathsf{A}, \bot_\mathsf{A}, \top_\mathsf{A}, \sqcup_\mathsf{A}, \sqcap_\mathsf{A} \rangle$ by a concretization function γ and an abstraction function α. In order to obtain a sound analysis, we require that they form a Galois connection; we denote it by $\langle \mathsf{D}, \sqsubseteq_\mathsf{C}, \bot_\mathsf{C}, \top_\mathsf{C}, \sqcup_\mathsf{C}, \sqcap_\mathsf{C} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathsf{D}^\#, \sqsubseteq_\mathsf{A}, \bot_\mathsf{A}, \top_\mathsf{A}, \sqcup_\mathsf{A}, \sqcap_\mathsf{A} \rangle$.
Since the most part of abstract domains does not satisfy the ascending chain condition, a widening operator ∇_A is required. This is an upper bound operator such that for all increasing chains $\mathsf{a}_0^\# \sqsubseteq_\mathsf{A} \ldots \mathsf{a}_n^\# \sqsubseteq_\mathsf{A} \ldots$ the increasing chain defined as $\mathsf{w}_0^\# = \mathsf{a}_0^\#, \ldots \mathsf{w}_{i+1}^\# = \mathsf{w}_i^\# \nabla_\mathsf{A} \mathsf{a}_{i+1}^\#$ is not strictly increasing.
Finally, the abstract semantics $\mathbb{S}^\#$ has to soundly approximate the concrete one, i.e. $\forall \mathsf{d}^\# \in \mathsf{D}^\# : \alpha \circ \mathbb{S}[\![\gamma(\mathsf{d}^\#)]\!] \sqsubseteq_\mathsf{A} \mathbb{S}^\#[\![\mathsf{d}^\#]\!]$.

1.3. Notation

Let S be a generic set. We denote by $\mathsf{S}^{\vec{+}}$ the set of all the finite traces composed by elements in S. A trace is an ordered sequence of elements. We denote by $len : [\mathsf{S}^{\vec{+}} \to \mathbb{N}]$ the function that given a trace returns its length, i.e. the number of elements contained in it. We denote by $istate : [(\mathsf{S}^{\vec{+}} \times \mathbb{N}) \to \mathbb{N}]$ the function that returns the i-th element of the given trace, where i is the integer value passed to the function.
As already stated in the introduction, we will introduce two levels of abstraction, and consequently we will denote by

S and s concrete sets and elements, by $S^\#$ and $s^\#$ sets and elements of the first level of abstraction, and by S° and s° sets and elements of the second level. In the same way, we will denote by *func* a function on concrete sets, by $func^\#$ the corresponding function at the first level of abstraction, and $func^\circ$ at the second level.

2. Syntax and concrete semantics

In this section we present a syntax focused on the write and read operations on shared memory, and the augmented concrete domain and semantics, in order to trace for each value the thread that wrote it in the shared memory.

2.1. Syntax

For the sake of readability, we consider a very restricted syntax, focused on the interactions with a shared memory, i.e. the read and the write actions. In this way we consider only statements of the form `sh_var = value` and `read(sh_var)`.
The statement `sh_var = value` writes on a shared variable `sh_var` a given value, while `read(sh_var)` reads a shared variable `sh_var` and returns its value. Note that this syntax can be easily extended with other statements (`if`, `while`, etc..).

2.2. Concrete domain

The concrete domain we consider is strictly focused on the shared memory. For each value, we trace the thread that wrote it. Since we are in the concrete context, at each point we know exactly which thread wrote a value, and we may relate each shared variable to a pair composed by its value and an identifier of that thread.
Formally, let T be the set of the identifiers of threads, Var be the set of shared variables, and V be the set of concrete values. The shared memory is a function that relates each variable to a pair composed by its value and the identifier of the thread that wrote it ($\mathsf{S} : [\mathsf{Var} \to (\mathsf{V} \times \mathsf{T})]$). Note that this representation of shared memory is quite different with respect to the approach usually adopted by current programming languages (e.g. Java), where the shared memory is the heap, and the values are accessed by reference. At static level, this approach would lead to multiple issues orthogonal w.r.t. our goal, in particular alias analysis, that deserve to be considered separately [6].
The idea behind a multithreaded program is to execute each thread in parallel. The thread-partitioned trace domain is aimed at formalizing this concept. The same approach has been adopted in previous works [7, 6].
The thread-partitioned domain represents an execution as a function that, given a thread, returns the trace containing only the shared memory locations obtained by its computation ($\Psi : [\mathsf{T} \to \mathsf{S}^{\vec{+}}]$). Each trace collects the execution of the thread to which it is related. All the possible executions of a program are represented by a set of these functions. As usual in abstract interpretation, the concrete domain is the lattice obtained using powerset operators $\langle \wp(\Psi), \subseteq, \emptyset, \wp(\Psi), \cup, \cap \rangle$.

2.3. Transfer function

In order to define the concrete semantics, we first introduce the $\mathbb{W} : [(\mathsf{S} \times \mathsf{T}) \to \mathsf{S}]$ of `sh_var = value` defined as $\mathbb{W}[\![\texttt{sh\_var = value}]\!](s,t) = s[\texttt{sh\_var} \mapsto (\mathsf{value}, t)]$.
The value of the shared variable when evaluating a shared memory s is given by the function $\mathbb{R} : [\mathsf{S} \to \mathsf{V}]$ of `eval(sh_var)` defined as $\mathbb{R}[\![\texttt{eval(sh\_var)}]\!](s) = \pi_1(s(\texttt{sh\_var}))$ where π_1 is the projection on the first component.
These two functions may be used in order to define a transfer function for a more complex language; usually in the abstract interpretation framework such a function is the basic step in order to define the concrete semantics as the result of fixpoint computation.

2.4. The running example

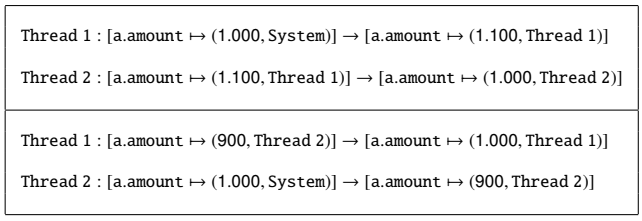


Figure 2. The concrete semantics

Figure 2 presents the two multithreaded concrete executions that can be obtained using the domain and the semantics just introduced. The first multithreaded trace represents the execution in which `Thread 1` is executed before `Thread 2`, while the second state depicts the opposite situation. Note that other executions are not possible, as both the methods are synchronized on the same monitor.

3. A value for each thread (abstraction 1)

In this section we present the first level of abstraction, where each shared variable is related to an abstract value for each thread that may write on that. Our analysis is parameterized on abstract domain that approximates numerical val-

ues (e.g. sign domain [3] that traces if they are positive or negative).

3.1. Abstract domain (first level)

We consider the following Galois connection between the concrete domain of values and its abstract counterpart:

$$\langle \wp(\mathsf{V}), \subseteq, \emptyset, \wp(\mathsf{V}), \cup, \cap \rangle \underset{\alpha_\mathsf{V}}{\overset{\gamma_\mathsf{V}}{\leftrightarrows}} \langle \mathsf{V}^\#, \sqsubseteq_{\mathsf{V}^\#}, \bot_{\mathsf{V}^\#}, \top_{\mathsf{V}^\#}, \sqcup_{\mathsf{V}^\#}, \sqcap_{\mathsf{V}^\#} \rangle$$

This means that the non-relational abstract domain for numerical values has to soundly approximate the concrete values. We also suppose that the identifiers of threads are the same in the concrete and in the abstract. This hypothesis may appear restrictive: current programming languages, like Java, identify threads by reference; the syntactic identification of threads is quite naive. On the other hand, we are presenting a new property on multithreaded programs; this property can be straightly extended also to a context on which threads are identified, for instance, by reference. In this case, it suffices to parameterize the analysis on the abstraction of references (e.g. a point-to analysis). For sake of simplicity, we suppose that threads are statically identified; on the other hand, this supposition still preserves the generality of our analysis. Moreover a previous work [6] has already proposed an alias analysis that is in position to statically identify Java threads.

In the concrete context, different executions may contain values written by different threads because of their random interleavings. This first level of abstraction traces a value for each thread; in this way it gathers all the values written by the same thread in the same abstract element.

Formally, the shared memory $\mathsf{s}^\#$ relates each shared variable to a function that maps a thread to the abstraction of the values that it may have written, i.e. $\mathsf{S}^\# : [\mathsf{Var} \to [\mathsf{T} \to \mathsf{V}^\#]]$.

As in the concrete semantics, we represent an execution as a function that, given a thread, returns the trace containing only the shared memories obtained by its computation $(\Psi^\# : [\mathsf{T} \to \mathsf{S}^{\#\vec{+}}])$, where $\mathsf{S}^{\#\vec{+}}$ denotes the set of all the finite traces with elements in $\mathsf{S}^\#$.

3.2. Upper bound operator

The upper bound operator on shared memories keeps all the values written by different threads; if there is a value related to the same thread in the two shared memories we make the join between these two values. With an abuse of notation, we suppose that if $\mathtt{var} \notin dom(\mathsf{s})$, then $dom(\mathsf{s}(\mathtt{var})) = \emptyset$.

$$\mathsf{s}_1^\# \sqcup_{\mathsf{S}^\#} \mathsf{s}_2^\# = \mathsf{s}^\# : \forall \mathtt{var} \in dom(\mathsf{s}_1^\#) \cup dom(\mathsf{s}_2^\#),\ \forall \mathsf{t} \in dom(\mathsf{s}_1^\#(\mathtt{var})) \cup dom(\mathsf{s}_2^\#(\mathtt{var})),$$

$$\mathsf{s}^\#(\mathtt{var})(\mathsf{t}) = \begin{cases} \mathsf{s}_1^\#(\mathtt{var})(\mathsf{t}) & \text{if } \mathtt{var} \notin dom(\mathsf{s}_2^\#) \vee \mathsf{t} \notin dom(\mathsf{s}_2^\#(\mathtt{var})) \\ \mathsf{s}_2^\#(\mathtt{var})(\mathsf{t}) & \text{if } \mathtt{var} \notin dom(\mathsf{s}_1^\#) \vee \mathsf{t} \notin dom(\mathsf{s}_1^\#(\mathtt{var})) \\ \mathsf{s}_1^\#(\mathtt{var})(\mathsf{t}) \sqcup_{\mathsf{V}^\#} \mathsf{s}_2^\#(\mathtt{var})(\mathsf{t}) & \text{otherwise} \end{cases}$$

The upper bound operator on traces simply applies the upper bound operator of shared memories on all the elements of all the traces of the two states.

$$\tau_1^\# \sqcup_{\mathsf{S}^{\#\vec{+}}} \tau_2^\# = \sigma_0^\# \to .. \to \sigma_i^\# : i = \max(len(\tau_1^\#), len(\tau_2^\#)),$$

$$\forall j \in [0..i] : \sigma_j^\# = \begin{cases} istate(\tau_1^\#, j) & \text{if } j < len(\tau_1^\#) \wedge \\ \quad \sqcup_{\mathsf{S}^\#} istate(\tau_2^\#, j) & \quad j < len(\tau_2^\#) \\ istate(\tau_2^\#, j) & \text{if } j \geq len(\tau_1^\#) \\ istate(\tau_1^\#, j) & \text{if } j \geq len(\tau_2^\#) \end{cases}$$

The upper bound operator on the multithreaded state is the pointwise extension of the upper bound of traces on all the elements of the codomain.

$$\mathsf{f}_1^\# \sqcup_{\Psi^\#} \mathsf{f}_2^\# = \mathsf{f}^\# : \forall \mathsf{t} \in dom(\mathsf{f}_1^\#) \cup dom(\mathsf{f}_2^\#) :$$

$$\mathsf{f}^\#(\mathsf{t}) = \begin{cases} \mathsf{f}_1^\#(\mathsf{t}) \sqcup_{\mathsf{S}^{\#\vec{+}}} \mathsf{f}_2^\#(\mathsf{t}) & \text{if } \mathsf{t} \in dom(\mathsf{f}_1^\#) \cap dom(\mathsf{f}_2^\#) \\ \mathsf{f}_1^\#(\mathsf{t}) & \text{if } \mathsf{t} \in dom(\mathsf{f}_1^\#) \wedge \mathsf{t} \notin dom(\mathsf{f}_2^\#) \\ \mathsf{f}_2^\#(\mathsf{t}) & \text{if } \mathsf{t} \in dom(\mathsf{f}_2^\#) \wedge \mathsf{t} \notin dom(\mathsf{f}_1^\#) \end{cases}$$

3.3. Abstraction function

The abstraction function maps a set of concrete shared memories into an abstract memory that relates each variable to a function that associates each thread with the abstraction of the values it may have written.

$$\alpha_\mathsf{S} : [\wp(\mathsf{S}) \to \mathsf{S}^\#]$$
$$\alpha_\mathsf{S}(\mathsf{S}) = \mathsf{f}^\# : \forall \mathtt{var} \in \bigcup_{\mathsf{s} \in \mathsf{S}} dom(\mathsf{s}), \forall \mathsf{t} \in \bigcup_{\mathsf{s} \in \mathsf{S}} dom(\mathsf{s}(\mathtt{var})) :$$
$$\mathsf{f}^\#(\mathtt{var})(\mathsf{t}) = \alpha_\mathsf{V}(\{\mathsf{v} : \exists \mathsf{s} \in \mathsf{S} : \mathsf{s}(\mathtt{var}) = (\mathsf{v}, \mathsf{t})\})$$

The abstraction of a set of traces produces an abstract trace such that its i-th element is the abstraction of the i-th elements of all the given concrete traces.

$$\alpha_{\mathsf{S}^\sharp} : [\wp(\mathsf{S}^{\vec{+}}) \to \mathsf{S}^{\#\vec{+}}]$$
$$\alpha_{\mathsf{S}^\sharp}(\mathsf{T}) = \sigma_0^\# \to \cdots \to \sigma_i^\# : i = \max(\textstyle\bigcup_{\tau \in \mathsf{T}} len(\tau)),$$
$$\forall j \in [0..i] : \sigma_j^\# = \alpha_\mathsf{S}(\textstyle\bigcup_{\tau \in \mathsf{T} : len(\tau) > j} istate(\tau, j))$$

When considering traces, the abstraction function returns an unique function that abstracts together all the traces produced by the same thread in the different executions.

$$\alpha_\Psi : [\wp(\Psi) \to \Psi^\#]$$
$$\alpha_\Psi(\Phi) = \mathsf{f}^\# : \forall \mathsf{t} \in \bigcup_{\mathsf{f} \in \Phi} dom(\mathsf{f}) :$$
$$\mathsf{f}^\#(\mathsf{t}) = \alpha_{\mathsf{S}^\sharp}(\textstyle\bigcup_{\mathsf{f} \in \Phi : \mathsf{t} \in dom(\mathsf{f})} \mathsf{f}(\mathsf{t}))$$

Theorem 1 (Soundness of $\langle \Psi^\#, \sqsubseteq_{\Psi^\#} \rangle$)

$$\langle \wp(\Psi), \subseteq \rangle \underset{\alpha_\Psi}{\overset{\gamma_\Psi}{\leftrightarrows}} \langle \Psi^\#, \sqsubseteq_{\Psi^\#} \rangle$$

3.4. Transfer function

In correspondence to the semantic functions $\mathbb{W}$ and $\mathbb{R}$, we introduce their abstract counterpart $\mathbb{W}^\#$ and $\mathbb{R}^\#$, as follows.

$\mathbb{W}^\# : [(\mathsf{S}^\# \times \mathsf{T}) \to \mathsf{S}^\#]$ of $\texttt{sh\_var} = \texttt{value}$ is defined as

$$\mathbb{W}^\#[\![\texttt{sh\_var} = \texttt{value}]\!](\mathsf{s}^\#, \mathsf{t}) = \\ = \mathsf{s}^\#[\texttt{sh\_var} \mapsto [\mathsf{t} \mapsto \alpha_{\mathsf{V}}(\texttt{value})]]$$

$\mathbb{R}^\# : [\mathsf{S}^\# \to \mathsf{V}^\#]$ of $\texttt{eval}(\texttt{sh\_var})$ is defined as

$$\mathbb{R}^\#[\![\texttt{eval}(\texttt{sh\_var})]\!](\mathsf{s}^\#) = \bigsqcup_{\mathsf{t} \in dom(\mathsf{s}^\#(\texttt{sh\_var}))} \mathsf{s}^\#(\texttt{sh\_var})(\mathsf{t})$$

As in the concrete semantics, these two functions may be used in order to define a transfer function for a more complex language.

Lemma 1 (Soundness) *$\mathbb{W}^\#$ and $\mathbb{R}^\#$ are respectively the abstraction of $\mathbb{W}$ and $\mathbb{R}$.*

3.5. The running example

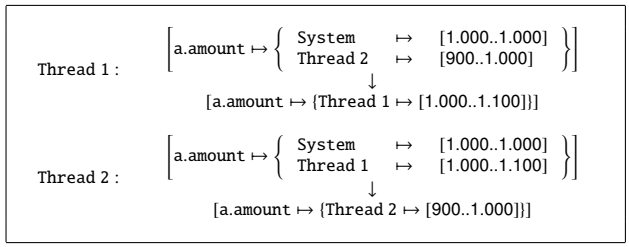


Figure 3. The abstract semantics

Figure 3 depicts the abstract semantics obtained by using the domain and the semantics just introduced. In order to capture numerical information we use the interval domain [3], where [a..b] represents the set of concrete values S such that $\forall \mathsf{n} \in \mathsf{S} : \mathsf{a} \leq \mathsf{n} \leq \mathsf{b}$.

4. Just one value (abstraction 2)

In this section we present the second level of abstraction, in which the values written by different threads collapse in the same abstract element.

4.1. Abstract domain (second level)

A shared memory $\mathsf{s}^\circ \in \mathsf{S}^\circ$ relates each variable var to a pair composed by a value and the set of threads that may have written it in var, formally $\mathsf{S}^\circ : [\mathsf{Var} \to (\mathsf{V}^\# \times \wp(\mathsf{T}))]$. We represent an execution as a function $\mathsf{f}^\circ \in \mathsf{S}^\circ$ that, given a thread, returns the trace containing only the shared memories obtained by its computation ($\Psi^\circ : [\mathsf{T} \to \mathsf{S}^{\circ\vec{\sharp}}]$).

4.2. Upper bound operator

The upper bound operator on shared memories makes the upper bound between values and the set union between the sets of threads that may have written the given value.

$$\mathsf{s}_1^\circ \sqcup_{\mathsf{S}^\circ} \mathsf{s}_2^\circ = \mathsf{s}^\circ : \forall \texttt{var} \in dom(\mathsf{s}_1^\circ) \cup dom(\mathsf{s}_2^\circ) : \mathsf{s}^\circ(\texttt{var}) = \\ = \begin{cases} (\pi_1(\mathsf{s}_1^\circ(\texttt{var})) \sqcup_{\mathsf{V}^\#} \pi_1(\mathsf{s}_2^\circ(\texttt{var})), & \text{if } \texttt{var} \in dom(\mathsf{s}_1^\circ) \cap \\ \quad \pi_2(\mathsf{s}_1^\circ(\texttt{var})) \cup \pi_2(\mathsf{s}_2^\circ(\texttt{var}))) & dom(\mathsf{s}_2^\circ) \\ \mathsf{s}_1^\circ(\texttt{var}) \text{ if } \texttt{var} \in dom(\mathsf{s}_1^\circ) \wedge \texttt{var} \notin dom(\mathsf{s}_2^\circ) \\ \mathsf{s}_2^\circ(\texttt{var}) \text{ if } \texttt{var} \in dom(\mathsf{s}_2^\circ) \wedge \texttt{var} \notin dom(\mathsf{s}_1^\circ) \end{cases}$$

The upper bound operator on traces simply applies the upper bound operator of shared memories on all the elements of all the traces of the two states.

$$\tau_1^\circ \sqcup_{\mathsf{S}^{\circ\vec{\sharp}}} \tau_2^\circ = \sigma_0^\circ \to .. \to \sigma_i^\circ : i = \max(len(\tau_1^\circ), len(\tau_2^\circ)), \\ \forall j \in [0..i] : \sigma_j^\circ = \begin{cases} istate(\tau_1^\circ, j) \sqcup_{\mathsf{S}^\circ} istate(\tau_2^\circ, j) & \text{if } j < len(\tau_1^\circ) \wedge j < len(\tau_1^\circ) \\ istate(\tau_1^\circ, j) & \text{if } j < len(\tau_1^\circ) \wedge j \geq len(\tau_2^\circ) \\ istate(\tau_2^\circ, j) & \text{if } j < len(\tau_2^\circ) \wedge j \geq len(\tau_1^\circ) \end{cases}$$

The upper bound operator on the multithreaded state is the pointwise extension of the upper bound of traces on all the elements of the codomain.

$$\mathsf{f}_1^\circ \sqcup_{\Psi^\circ} \mathsf{f}_2^\circ = \mathsf{f}^\circ : \forall \mathsf{t} \in dom(\mathsf{f}_1^\circ) \cup dom(\mathsf{f}_2^\circ) : \\ \mathsf{f}^\circ(\mathsf{t}) = \begin{cases} \mathsf{f}_1^\circ(\mathsf{t}) \sqcup_{\mathsf{S}^{\circ\vec{\sharp}}} \mathsf{f}_2^\circ(\mathsf{t}) & \text{if } \mathsf{t} \in dom(\mathsf{f}_1^\circ) \cap dom(\mathsf{f}_2^\circ) \\ \mathsf{f}_1^\circ(\mathsf{t}) & \text{if } \mathsf{t} \in dom(\mathsf{f}_1^\circ) \wedge \mathsf{t} \notin dom(\mathsf{f}_2^\circ) \\ \mathsf{f}_2^\circ(\mathsf{t}) & \text{if } \mathsf{t} \in dom(\mathsf{f}_2^\circ) \wedge \mathsf{t} \notin dom(\mathsf{f}_1^\circ) \end{cases}$$

4.3. Abstraction function

The abstraction function abstracts an element of the first level of abstraction.

$$\alpha_{\mathsf{S}^\#} : [\mathsf{S}^\# \to \mathsf{S}^\circ] \\ \alpha_{\mathsf{S}^\#}(\mathsf{s}^\#) = \mathsf{s}^\circ : \forall \texttt{var} \in dom(\mathsf{s}^\#) : \\ \mathsf{s}^\circ(\texttt{var}) = (\bigsqcup_{\mathsf{t} \in dom(\mathsf{s}^\#(\texttt{var}))} \mathsf{s}^\#(\mathsf{t}), dom(\mathsf{s}^\#(\texttt{var})))$$

The abstraction of traces is the pointwise application of the abstraction of the shared memory to all the elements of all the traces.

$$\alpha_{\mathsf{S}^{\#\vec{\sharp}}} : [\mathsf{S}^{\#\vec{\#}} \to \mathsf{S}^{\circ\vec{\sharp}}] \\ \alpha_{\mathsf{S}^{\#\vec{\sharp}}}(\sigma_0^\# \to .. \to \sigma_i^\#) = \alpha_{\mathsf{S}^\#}(\sigma_0^\#) \to .. \to \alpha_{\mathsf{S}^\#}(\sigma_i^\#)$$

The abstraction on the multithreaded state is the pointwise application of the abstraction of traces.

$$\alpha_{\Psi^\#} : [\Psi^\# \to \Psi^\circ] \\ \alpha_{\Psi^\#}(\mathsf{f}^\#) = \mathsf{f}^\circ : \forall \mathsf{t} \in dom(\mathsf{f}^\#) : \mathsf{f}^\circ(\mathsf{t}) = \alpha_{\mathsf{S}^{\#\vec{\sharp}}}(\mathsf{f}^\#(\mathsf{t}))$$

Theorem 2 (Soundness of $\langle \Psi^\circ, \sqsubseteq_{\Psi^\circ} \rangle$)

$$\langle \Psi^\#, \sqsubseteq_{\Psi^\#} \rangle \underset{\alpha_{\Psi^\#}}{\overset{\gamma_{\Psi^\#}}{\leftrightarrows}} \langle \Psi^\circ, \sqsubseteq_{\Psi^\circ} \rangle$$

4.4. Transfer function

The transfer function $\mathbb{W}^\circ : [(\mathsf{S}^\circ \times \mathsf{T}) \rightarrow \mathsf{S}^\circ]$ of $\mathtt{sh\_var} = \mathtt{value}$ is defined as

$$\mathbb{W}^\circ[\![\mathtt{sh\_var} = \mathtt{value}]\!](\mathsf{s}^\circ, \mathsf{t}) = \mathsf{s}^\circ[\mathtt{sh\_var} \mapsto (\mathsf{value}^\#, \{\mathsf{t}\})]$$

Indeed the function $\mathbb{R}^\circ : [\mathsf{S}^\circ \rightarrow \mathsf{V}^\#]$ of $\mathtt{eval(sh\_var)}$ is defined as

$$\mathbb{R}^\circ[\![\mathtt{eval(sh\_var)}]\!](\mathsf{s}^\circ) = \pi_1(\mathsf{s}^\circ(\mathtt{sh\_var}))$$

Lemma 2 (Soundness) *$\mathbb{W}^\circ$ and $\mathbb{R}^\circ$ are respectively the abstraction of $\mathbb{W}^\#$ and $\mathbb{R}^\#$.*

4.5. The running example

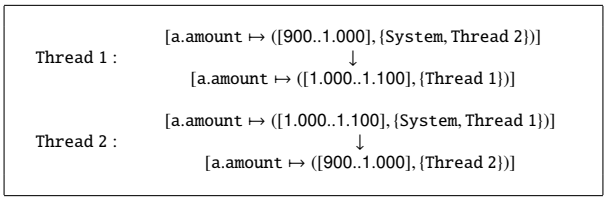


Figure 4. The abstract semantics on the running example

Figure 4 presents the abstract semantics obtained using the domain and the semantics just introduced. Also in this case we use the interval domain in order to capture numerical information.

5. The deterministic property

In the last three sections, we have presented the concrete domain and semantics and the two levels of abstraction. In this section we define on them the deterministic property.

5.1. Determinism

A program is generally said to be deterministic "if given an input it returns always the same output"[5]. This definition does not specify what is the input and the output of a program. Someone may think that the output is what it writes on the screen, some others that it is the state of the memory at the end of the execution, or during it.
In a multithreaded context, a too restrictive application of the concept of determinism may lead to define that a multithreaded program is deterministic if all the statements, even of different threads, are executed always in the same total order.
We face this problem through the thread-partitioned domain, that abstracts away the inter-thread order in which the statements are executed. At this level a program is deterministic if, given an input, returns the same output for any order of execution.
Moreover our work is focused on the non-determinism induced by the random interleavings of the execution of different threads. As we do not make any supposition or restriction on the programming language and its semantics, many different forms of non-determinism may exist: a random number produced by the system, and the inputs received by an operator are two typical examples of non-determinism not induced by the multithreaded execution. We want to catch and analyze only the non-deterministic behaviors due to the parallel executions of threads, discarding all the other cases.
This goal can be reached by using the information collected by our analysis, as we trace for each value on the shared memory which thread may have written it: it suffices to check if at a given point (i.e. for a given state of execution of a given thread) a value could have been written by two different threads.

5.2. Formal definition of determinism on the concrete domain

We first define the determinism as difference between two states, and then we apply it to a set of elements of the concrete multithreaded semantics.

Definition 1 (Determinism on shared memory) *Given two states, they underline a non deterministic behavior due to the multithreaded execution if the two states relate the same variable to values written by different threads.*

$$\begin{gathered} DS : [\mathsf{S} \times \mathsf{S} \rightarrow \{\mathsf{true}, \mathsf{false}\}] \\ DS(\mathsf{s}_1, \mathsf{s}_2) = \mathsf{false} \\ \Updownarrow \\ \exists \mathtt{var} \in dom(\mathsf{s}_1) \cap dom(\mathsf{s}_2) : \mathsf{s}_1(\mathtt{var}) = (\mathsf{val}_1, \mathsf{t}_1), \\ \mathsf{s}_2(\mathtt{var}) = (\mathsf{val}_2, \mathsf{t}_2), \mathsf{t}_1 \neq \mathsf{t}_2 \end{gathered}$$

Definition 2 (Determinism on multithreaded state)

$$\begin{gathered} D : [\wp(\Psi) \rightarrow \{\mathsf{true}, \mathsf{false}\}] \\ D(\theta) = \mathsf{false} \\ \Updownarrow \\ \exists \mathsf{f}_1, \mathsf{f}_2 \in \theta : \exists \mathsf{t} \in dom(\mathsf{f}_1) \cap dom(\mathsf{f}_2) : \\ \tau_1 = \mathsf{f}_1(\mathsf{t}), \tau_2 = \mathsf{f}_2(\mathsf{t}), \exists i \in [0.. \min(len(\tau_1), len(\tau_2))] : \\ DS(istate(\tau_1, i), istate(\tau_2, i)) = \mathsf{false} \end{gathered}$$

5.3. First level of abstraction

$$DS^{\#} : [\mathsf{S}^{\#} \to \{\mathsf{true}, \mathsf{false}\}]$$
$$DS^{\#}(\mathsf{s}^{\#}) = \mathsf{false}$$
$$\Updownarrow$$
$$\exists \mathtt{var} \in dom(\mathsf{s}^{\#}) : |dom(\mathsf{s}^{\#}(\mathtt{var}))| > 1$$

$$D^{\#} : [\Psi^{\#} \to \{\mathsf{true}, \mathsf{false}\}]$$
$$D^{\#}(\mathsf{f}^{\#}) = \mathsf{false}$$
$$\Updownarrow$$
$$\exists \mathsf{t} \in dom(\mathsf{f}^{\#}) : \tau^{\#} = \mathsf{f}^{\#}(\mathsf{t}), \exists i \in [0..len(\tau^{\#})] :$$
$$DS^{\#}(istate(\tau^{\#}, i)) = \mathsf{false}$$

Lemma 3 (Soundness) *$D^{\#}$is sound w.r.t. D, i.e.:*

$$\forall \theta \in \wp(\Psi) : D(\theta) = \mathsf{false} \Rightarrow D^{\#}(\alpha_{\Psi}(\theta)) = \mathsf{false}$$

5.4. Second level of abstraction

$$DS^{\circ} : [\mathsf{S}^{\circ} \to \{\mathsf{true}, \mathsf{false}\}]$$
$$DS^{\circ}(\mathsf{s}^{\circ}) = \mathsf{false} \Leftrightarrow \exists \mathtt{var} \in dom(\mathsf{s}^{\circ}) : |\pi_2(\mathsf{s}^{\circ}(\mathtt{var}))| > 1$$

$$D^{\circ} : [\Psi^{\circ} \to \{\mathsf{true}, \mathsf{false}\}]$$
$$D^{\circ}(\mathsf{f}^{\circ}) = \mathsf{false}$$
$$\Updownarrow$$
$$\exists \mathsf{t} \in dom(\mathsf{f}^{\circ}) : \tau^{\circ} = \mathsf{f}^{\circ}(\mathsf{t}), \exists i \in [0..len(\tau^{\circ})] :$$
$$DS^{\circ}(istate(\tau^{\circ}, i)) = \mathsf{false}$$

Lemma 4 (Soundness) *$D^{\circ}$is sound w.r.t. $D^{\#}$, i.e.:*

$$\forall \mathsf{f}^{\#} \in \Psi^{\#} : D^{\#}(\mathsf{f}^{\#}) = \mathsf{false} \Rightarrow D^{\circ}(\alpha_{\Psi^{\#}}(\mathsf{f}^{\#})) = \mathsf{false}$$

5.5. The running example

The deterministic property on the concrete semantics discovers that the running example introduced by Section 1.1 does not respect it. In particular, the two executions differ on the values read by both the threads: in the first one, `Thread 1` reads a value written by `System` and `Thread 2` the one written by `Thread 2`, while in the second execution `Thread 1` reads the value of `Thread 2` and `Thread 2` the one of `System`. In Figure 5 we underline the identifiers of threads in the cases in which a non-deterministic behavior arises.
Since the two level of abstractions are sound w.r.t. determinism, they do not validate the deterministic property.

6. Weak determinism

In this section we introduce a new concept of determinism which is weaker than the previous definitions and that is defined on the first level of abstraction.

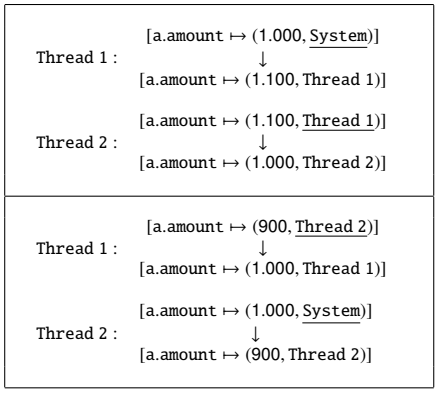


Figure 5. The non-deterministic behaviors in the concrete semantics

6.1. Approximating numerical values

The first level of abstraction is parameterized on a non-relational abstract domain that approximates the numerical values. Moreover, we collect for each thread an abstract value that approximates all the concrete values it may have written at that point. Through these two observations we are in position to define a new property: the weak determinism. The idea is that two different concrete values do not produce different observable behaviors if their abstraction is the same. This concept has to be tuned on an abstract domain: for instance, with the sign domain it would mean that at a given point all the values written in parallel on a given variable of the shared memory have the same sign.

6.2. Formal definition

Definition 3 (Weak determinism on shared memory)

$$ADS^{\#} \quad : [\mathsf{S}^{\#} \to \{\mathsf{true}, \mathsf{false}\}]$$
$$ADS^{\#} \quad (\mathsf{s}^{\#}) = \mathsf{false}$$
$$\Updownarrow$$
$$\exists \mathtt{var} \in dom(\mathsf{s}^{\#}) : |dom(\mathsf{s}^{\#}(\mathtt{var}))| > 1 \wedge$$
$$\exists \mathsf{t}_1, \mathsf{t}_2 \in dom(\mathsf{s}^{\#}(\mathtt{var})) : \mathsf{s}^{\#}(\mathtt{var})(\mathsf{t}_1) \neq \mathsf{s}^{\#}(\mathtt{var})(\mathsf{t}_2)$$

Definition 4 (Weak determinism on multithreaded state)

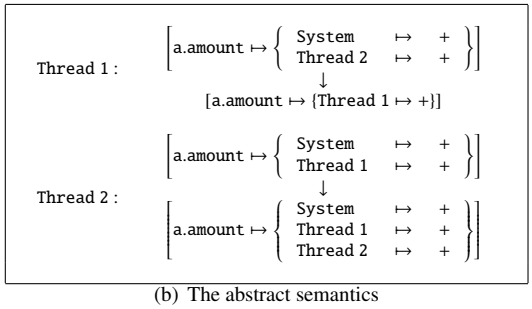
$$AD^{\#} \quad : [\Psi^{\#} \to \{\mathsf{true}, \mathsf{false}\}]$$
$$AD^{\#} \quad (\mathsf{f}^{\#}) = \mathsf{false}$$
$$\Updownarrow$$
$$\exists \mathsf{t} \in dom(\mathsf{f}^{\#}) : \tau^{\#} = \mathsf{f}^{\#}(\mathsf{t}), \exists i \in [0..len(\tau^{\#})] :$$
$$AD^{\#}(istate(\tau^{\#}, i)) = \mathsf{false}$$

Note that, even if a program is not deterministic in the concrete, the weak determinism may be validated it, as the values written by different threads may produce the same abstract element.

6.3. Example 2

```
public void withdraw2(int money) {
 synchronized(this) {
  int temp=account.amount-money;
  if(temp>0)
   account.amount=temp;
 }
}
```

(a) The source code



(b) The abstract semantics

Figure 6. Example 2

Suppose we now modify the `withdraw` method presented in section 1.1 as depicted in Figure 6.(a).
This method allows the withdrawing only if there is enough money in the bank account. Suppose we analyze the multi-threaded program presented by the running example when replacing the `withdraw` method with this one. We use the sign domain to capture numerical information; in this way we capture for each numerical value if it is positive (+), negative (−) or equal to zero (0). The result of the analysis at the first level of abstraction is depicted by Figure 6.(b); we simplify the elements depicting just the results of the `withdraw` method, ignoring the internal computation of the thread.
Applying the weak deterministic property on that abstract value we obtain that such a property is validated: in fact in all the possible executions the value stored by the field `amount` is always positive. On the other side, the full determinism is not guaranteed, as the amount of money may have been written in parallel by different threads.

7. From determinism to semi-automatic parallelization

In this section we sketch how the determinism of a program may be useful in order to semi-automatically parallelize it.

7.1. Motivation

As already explained by the introduction, multi-core architectures have recently appeared in a broad market, and the only way to take advantage from this technology is to develop applications that perform as many tasks in parallel as possible. On the other hand, reasoning about parallel applications is strictly more difficult than on sequential programs. The logical consequence is that (semi-) automatic tools in order to find and detect possible parallelizable fragments of sequential code are particularly useful.

7.2. Determinism and parallelism

By definition, a sequential program performs a set of sequential operations. Usually, an operation reads some data written or modified by previous statements and it writes values that will be used during the rest of the computation. When each sequential operation deals with a disjointed set of variables, the program is trivially parallelizable. However, the latter condition does not apply to the most part of sequential programs.
Through our deterministic property, given two subsets of a sequential program, we can analyze them as if they were executed in parallel, and check if there are some non-deterministic behaviors due to the parallel execution. If it is not the case, the two parts of the sequential program can run in parallel without inducing any new behavior because of the parallel execution.

7.3. Relaxing the deterministic property

In order to find parallelizable blocks in a program, we need also to relax the deterministic property in order to focus only on the critical parts of the program. A first approach consists in restricting the check of the deterministic property only: (i) on the traces of some threads (e.g. only the ones that performs critical operations); (ii) on a subset of the shared memory's variables (e.g. only the amount of money of a bank account, and not the name of the owner); (iii) on a subset of the states (e.g. only the ones that write on the shared memory). A priori, we cannot know which components may be ignored, and this should be an input by the developer, as he is the only one that knows which part are critical and which are not.
Another approach is to use the weak determinism presented by Section 6.
These two approaches may be combined together. For instance, we can express something like "check the weak determinism through the sign domain on the amount of money".

8. Experimental results

The analyzer presented by [6] (that works on Java multi-threaded programs at bytecode level) has been extended in order to study the deterministic property; in addition, it has

Program	# St.	# Th.	An.	Det.	Weak Det.
philo	213	2	< 1”	< 1”	< 1”
tsp	1936	2	24”	< 1”	< 1”
elevator	1829	2	15”	< 1”	< 1”
barrier	363	3	< 1”	< 1”	< 1”
forkjoin	170	2	< 1”	< 1”	< 1”
sync	320	3	< 1”	< 1”	< 1”
sor	1121	2	4”	< 1”	< 1”
crypt	2636	3	4”	< 1”	< 1”
lufact	3732	2	26”	< 1”	< 1”
montecarlo	3864	2	46”	4”	4”
moldyn	9029	2	13’51”	50”	1’01”

Table 1. Experimental results

already been extended also to be sound w.r.t. the happens-before memory model, following the theoretical results of [7]. We tested the analysis of deterministic and weak deterministic properties on a set of well-known benchmarks: philo, tsp, and elevator are taken from [19], while the others (barrier, forkjoin, sync, sor, crypt, lufact, montecarlo, and moldyn) are taken from the Java Grande Forum Benchmark Suite [1]. All tests have been performed on an AMD Athlon 64 3000+ processor with 2 GB of RAM running a GNU/Linux operating system with the Java virtual machine version 1.6.0_06-b02.

Table 1 depicts the experimental results. The first column reports the name of the program, while the following two columns report respectively the number of bytecode statements and of abstract threads analyzed. The fourth column depicts the time required in order to build up an abstraction of the program following the approach introduced by [7], while the last two columns report respectively the time spent to analyze the deterministic and the weak-deterministic property on the results obtained by this abstraction.

All the programs contain a potentially unbounded number of concrete threads; these are abstracted into a finite set through an ad-hoc alias analysis (note that in Java threads are objects and so identified by reference) [6]. The time required in order to validate the properties is much less than the one required by the analysis in order to build up an abstraction of the program; in average, it requires the 6% of the overall computational time. In addition, in the worst case in order to analyze almost 10k of Java bytecode statements the overall analysis require about 15’. We think that these experimental results are particularly encouraging, considering also that the analysis has not been optimized at all, and we think we may easily obtain better computational times.

9. Related works

During the last decades, a topic largely debated by researchers has been how parallel threads can communicate in order to bound the nondeterministic behaviors. Many different approaches have been proposed, but none has been able to solve this problem.

A first way has been the static or dynamic checking on the actions performed by different threads, like general and data races [16]. The idea is that races cause nondeterminism, and then they are symptom of bugs. On one hand, the absence of general races is a too restrictive condition for multithreaded programs that communicate asynchronously through shared memory. On the other hand, avoiding data races requires sometimes unnecessary synchronization actions, and it does not guarantee at all the determinism of a program. In addition, a race does not take into account which statements and threads cause it, with which area of the shared memory it deals, if the informations written in parallel are different and how much. Our intuition is to directly study nondeterminism, and in this way we can relax the property on a critical subset of the shared memory, considering only some statements and threads, looking also at the (abstract) values written and read in parallel. In this way, our approach is strictly more flexible than general and data races.

Another big amount of work has been focused on the consistency conditions that multithreaded programs have to provide. The most known model is sequential consistency [12], that has been proved to be too restrictive for current programming language. For instance, the Java memory model [14] is more relaxed than it, and more restrictive than the happens-before memory model [11]. Our approach is orthogonal w.r.t. these consistency conditions, as they define which states of shared memory are visible during the execution of different threads. In addition a previous work [7] has already defined a static analysis sound w.r.t. the happens-before memory model, that is an over-approximation of the Java one.

A similar approach has been the definition of a specific model in order to reduce the non-determinism of multithreaded programs. The Software Transactional Memory (STM) [17, 9] allows to specify that the execution of a given part of program is atomic, i.e. it is viewed as an unique operation by other threads. A large work on the semantics of parallel languages has been developed by trace theory [15], as [8] that presents a deterministic semantics for a simple parallel programming language. In this case, our idea is to distinguish the model of execution and the determinism of a program, as they are two orthogonal issues. Our deterministic and weak deterministic prop-

erties may be applied to analyses sound w.r.t. sequential consistency, happens-before memory model, Java memory model, software transactional memory, etc etc..

10. Conclusion and future work

In this paper, we show how to define and statically analyze the determinism of multithreaded programs. The main novelty w.r.t. previous approaches is that our work deals directly with the determinism, and in particular with the behaviors due to the multithreaded executions, instead of, for instance, defining a restricted model. In this way we are able to define and statically analyze which programs are deterministic w.r.t. their multithreaded executions and which are not. In addition, since we are dealing at the source of the problem, our approach may lead to new and unexplored ideas, as the weak determinism and the semi-automatic parallelization of sequential programs. The soundness of our analysis has been formally proved, and we presented some experimental results when applying the analysis to a set of well-known benchmarks.
As future works, we want to investigate how to apply this property in order to propose to the developers possible parallelizations of sequential programs. In particular, we may rely on the information obtained through the cost analysis [2] in order to detect which sequential statements are computationally heavier, and to check if their parallelization induces non-deterministic behaviors.

References

[1] Java Grande Forum Benchmark Suite. At http://www.epcc.ed.ac.uk/research/activities/java-grande/.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In Springer, editor, *Proceedings of ESOP '07*, LNCS, 2007.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In ACM Press, editor, *Proceedings of POPL '77*, 1977.

[4] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In ACM Press, editor, *Proceedings of POPL '79*, 1979.

[5] Perry A. Emrath and David A. Padua. Automatic detection of nondeterminacy in parallel programs. In ACM Press, editor, *Proceedings of PADD '88*, 1988.

[6] P. Ferrara. A fast and precise analysis for data race detection. In Elsevier, editor, *Proceedings of Bytecode '08*, volume ENTCS, 2008.

[7] P. Ferrara. Static analysis via abstract interpretation of the happens-before memory model. In Springer, editor, *Proceedings of TAP '08*, LNCS, 2008.

[8] Paul Gastin and Michael W. Mislove. A truly concurrent semantics for a simple parallel programming language. In Springer, editor, *Proceedings of CSL '99*, 1999.

[9] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In ACM Press, editor, *Proceedings of PPoPP '05*, 2005.

[10] G. Koch. Discovering multi-core: extending the benefits of Moore's law. In *Technology Intel Magazine*. Intel, July 2005.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In ACM Press, editor, *Commun. ACM*, 1978.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. In *IEEE Trans. Computers*, 1979.

[13] Edward A. Lee. The problem with threads. In IEEE Computer Society Press, editor, *Computer*, 2006.

[14] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In ACM Press, editor, *Proceedings of POPL '05*, 2005.

[15] A Mazurkiewicz. Trace theory. In Springer, editor, *Advances in Petri nets 1986*, 1987.

[16] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. In ACM Press, editor, *ACM Lett. Program. Lang. Syst.*, 1992.

[17] Nir Shavit and Dan Touitou. Software transactional memory. In ACM Press, editor, *Symposium on Principles of Distributed Computing*, 1995.

[18] Herb Sutter and James Larus. Software and the concurrency revolution. In ACM Press, editor, *ACM Queue*, 2005.

[19] C. Von Praun and T. R. Gross. Object race detection. In ACM Press, editor, *Proceedings of OOPSLA 01*, 2001.