Causal Atomicity

Azadeh Farzan[⋆] P. Madhusudan
Department of Computer Science,
University of Illinois at Urbana-Champaign.
{afarzan,madhu}@cs.uiuc.edu

Abstract. Atomicity is an important generic specification that assures that a programmer can pretend blocks occur sequentially in any execution. We define a notion of atomicity based on *causality.* We model the control flow of a program with threads using a Petri net that naturally abstracts data, and faithfully captures the independence and interaction between threads. The causality between events in the partially ordered executions of the Petri net is used to define the notion of *causal atomicity.* We show that causal atomicity is a robust notion that many correct programs adopt, and show how we can effectively check causal atomicity using Petri net tools based on unfoldings, which exploit the concurrency in the net to yield automatic partial-order reduction in the state-space.

1 Introduction

Programs with multiple threads are a common paradigm in concurrent programming. Programming correctly with threads is particularly hard as one has to consider the various interleavings of threads at run-time. Moreover, bugs that manifest themselves because of interleavings are harder to detect using testing, as they can be very infrequent in occurrence. A practical approach to programming threads is to develop techniques that allow the programmer to specify and verify disciplined interaction between threads.

The lack of *race conditions* is such a discipline; a race condition occurs when two threads simultaneously access a global variable, and one of the accesses is a *write*. Depending on when the write event gets scheduled, the program could take very different paths, which is a cause of concern. While the lack of races does seem to be a natural discipline to adhere to, it has been observed and argued that it is not a strong enough condition [10].

A stronger[1] discipline is to require methods or blocks of code to be *atomic*. The general definition of atomicity is: *a code block is atomic [10] if for every interleaved execution of the program in which the code block is executed, there is an equivalent run of the program where the code block is executed sequentially (without interleaving with other threads).* Intuitively, since for every interleaved execution t, there is an equivalent execution t' where the block occurs sequentially,

⋆ Research supported by ONR grant N00014-02-1-0715.

[1] Atomicity is not a strictly stronger notion than race-freedom; see Figure 7 for an example.

if the block had a logical error making t incorrect, then t' would be incorrect as well, arguing that the error in the block was not because of its interleaving with other threads. Consequently, a block being atomic greatly simplifies writing and understanding code: the programmer can *pretend* that the code block is executed sequentially, which simplifies its correctness argument. Note that the problem of checking atomicity of program blocks is obviously undecidable.

Atomicity is a well-argued principle of programming: in the database literature, it is known as *serializability* [6, 19, 2, 1, 21], and in the software analysis literature, it has been argued that many errors in threaded code can be traced back to blocks being non-atomic [10, 9, 7, 23, 13, 8, 22]. There has been extensive work on detecting atomicity of program blocks: static analysis techniques that use type-checking to detect *transactions* [15, 10, 9, 23], where a transaction is a strong notion that implies atomicity; dynamic analysis that checks atomicity of tests of the program at run-time [8, 22]; and model checking for atomicity where a monitor that detects non-atomic blocks runs in parallel with the system, which is then model checked [13, 7].

In order to get effective algorithms, atomicity checkers aim for soundness (i.e. if the tool reports a block to be atomic, then the block should indeed be atomic), and the generic way to achieve this is to abstract the program in a sound fashion, as well as define a sound notion of *equivalence* between abstract traces. In other words, the equivalence relation between traces of the abstract model should imply that the concrete traces represented by them are equivalent as well.

While atomicity checkers in the literature do assure that their analyses are sound, they do not define precisely the abstraction they use, nor define precisely the notion of equivalence they assume. For example, static atomicity checking using types are based on transactions (transactions imply atomicity), but transactions require knowing what kind of "mover" each statement is, which is achieved using a separate race-checking analysis (which again is not precisely defined). The algorithm for checking for transactions is then implemented using types, and again it is not argued whether given the classification of statements as movers, the type-checking approach is *complete* or not. All in all, though every step is sound, and the soundness of the whole algorithm is assured, the precise abstraction and notion of equivalence used is not clear, making it hard to evaluate and compare the formalisms.

The main contribution of this paper is a new notion of atomicity for programs based on *causality*, and that has precise definitions of the abstraction mechanism and the notion of equivalence between abstract runs. Given a program P with multiple threads, we exhibit a clean *control* model of the program as a Petri net [18, 16]. This modeling is aimed at capturing control and abstracting away the data values used in the program. Moreover, the Petri net explicitly captures the *independence* of execution of threads, and the interaction mechanism of the threads (using shared variables, locks, etc.). The model for the program is independent of any notion of atomicity, and captures just the dependence and independence of control in the threads of the program.

This model of a Petri net generates, in a standard way, partially-ordered runs of P that depict possible control interactions between threads [4]; we call these partially-ordered runs the *control traces* of P. The partially ordered control traces depict the set of events that have occurred and also define the *causal* relation between these events (such a causal structure is not evident in the *linear* runs of P). Moreover, the Petri net model is such that if one linearization σ of a partially ordered run Tr is feasible in the original program, then all linearizations of the Tr are feasible in the concrete program as well and are equivalent to r (in terms of the final state reached).

Causal atomicity is defined using the causal structure of the control traces generated by the program. We consider two sequential executions of a program to be *equivalent* if and only if they correspond to linearizations of the same partially-ordered trace of the program. Causal atomicity reduces to a very simple property on partially ordered traces: a block B of a thread is causally atomic if there is no control trace of the program where an event of another thread occurs *causally after* the beginning of B and *causally before* another event that is within the same block B.

Our notion of causal atomicity is simple and yet powerful enough to capture common interaction disciplines in correct programs. Our notion is certainly stronger than looking for patterns of *transactions* [15, 10], and can handle programs that do not explicitly use locks.

Turning to algorithms for checking atomicity, we show how causal atomicity can be checked using *partial-order* techniques based on unfoldings of Petri nets. Our algorithm is sound *and* complete in checking for causal atomicity of the net. Given a Petri net model of P with a block marked to be checked for atomicity, we show how to reduce the problem of checking causal atomicity of P to a coverability problem for an associated *colored* Petri net Q [14]. The latter problem is decidable and there are several tools that can efficiently check coverability in colored Petri nets. In particular, the tools that use *unfolding* techniques [17, 5] of nets are useful as they exploit the structure of the net to give automatic reduction in state-space (akin to partial-order reduction that has been exploited in model checking concurrent systems).

Finally, we show that causal atomicity is a common paradigm in many programs by considering several examples. We report on experiments that reduce checking causal atomicity to coverability, where the latter is solved using the PEP tool (Programming Environment based on Petri nets) [12]. The experiments show that causal atomicity lends itself to fast partial-order based model checking.

When there is only one block that is being checked for atomicity, our notion of atomicity is the same as the notion of *serializability* studied for database transactions [11, 1]. However, when there are multiple blocks, serializability demands that for every execution, there is one execution where *all* the atomic blocks are executed serially, while our notion demands that for every execution and every block, there is some execution where *that* block occurs sequentially. We believe our notion is more appropriate for threaded software. Figure 1 shows an example of a trace of a program with four threads and two blocks which intuitively ought to be declared atomic. For instance, any pre-post condition of the block B (or B') that depends only on the variables used in the block holds on all interleaved runs provided it holds in runs where the block is executed sequentially. Note a program with such a trace would be declared non-serializable, but declared causally atomic.

While we believe the jury is still out on which of these notions of atomicity is useful and accurate for checking programs, note that our contributions hold equally well for serializability: we can define a notion of serializability using the causal edges in the Petri net model and check for it using unfolding algorithms (however, checking serializability seems more complex than checking causal atomicity).

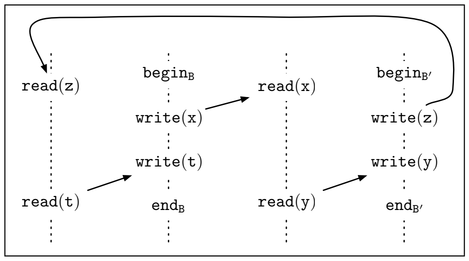


Fig. 1. Serializable but not causally atomic

The paper is structured as follows. Section 2 introduces a simple syntax for a programming language with threads, and defines Petri nets and the partially ordered traces they generate. Section 3 defines the modeling of a program as a Petri net and defines causal atomicity based on the traces generated by this net. Section 4 gives the generic translation of such a program model into a colored Petri net, reducing causal atomicity to coverability. Section 5 gives experimental results that show the efficacy of partial-order model checking tools in detecting causal atomicity, and Section 6 contains concluding remarks.

2 Preliminaries

2.1 The Language for Programs

We base our formal development on the language SML (Simple Multithreaded Language). Figure 2 presents the syntax of SML. The number of threads in an SML program is fixed and preset. There are two kinds of variables: local and global, respectively identified by the sets *LVar* and *GVar*. All variables that appear at the definition list of the program are global and shared among all threads. Any other variable that is used in a thread is assumed to be local to the thread.

We assume that all variables are integers and are initialized to zero. We use small letters (capital letters) to denote local (global, resp.) variables. *Lock* is a global set of locks that the threads can use for synchronization purposes through `acquire` and `release` primitives. The semantics of a program is the obvious one and we do not define it formally.

`begin` and `end` primitives are used to mark the beginning and end of a block that is intended to be checked for atomicity. The goal of the atomicity checker is to check whether all such blocks are indeed atomic. Figure 5 shows two examples of multithreaded programs written in SML.

$$\begin{array}{rll}
\text{P} ::= & \text{defn thlist} & \text{(program)}\\
\text{thlist} ::= & \text{null} \mid \text{stmt} \mid\mid \text{thlist} & \text{(thread list)}\\
\text{defn} ::= & \texttt{int}\ Y \mid \texttt{lock}\ l \mid \text{defn} ; \text{defn} & \text{(variable declaration)}\\
\text{stmt} ::= & x := e & \\
\mid & \texttt{while}\ (b)\ \{\ \text{stmt}\ \} \mid \texttt{begin}\ \text{stmt}\ \texttt{end} & \\
\mid & \texttt{if}\ (b)\ \{\ \text{stmt}\ \}\ \text{else}\ \{\ \text{stmt}\ \} \mid \texttt{skip} & \\
\mid & \texttt{acquire}(l) \mid \texttt{release}(l) \mid \text{stmt} ; \text{stmt} & \text{(statement)}\\
e ::= & i \mid x \mid Y \mid e+e \mid e*e \mid e/e & \text{(expression)}\\
b ::= & \texttt{true} \mid \texttt{false} \mid e\ op\ e \mid b \vee b \mid \neg b & \text{(boolean expression)}\\
 & & \\
 & op \in \{<, \leq, >, \geq, =, !=\} & \\
 & x \in \text{LVar},\ Y \in \text{GVar}, \quad i \in \text{Integer},\ l \in \text{Lock} &
\end{array}$$

Fig. 2. SML syntax

2.2 Petri Nets and Traces

Definition 1. *A Petri net is a triple $N = (P, T, F)$, where P is a set of places, T (disjoint from P) is a set of transitions, and $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation.*

For a transition t of a (Petri) net, let ${}^{\bullet}t = \{p \in P | (p,t) \in F\}$ denote its set of pre-conditions and $t^{\bullet} = \{p \in P | (t,p) \in F\}$ its set of post-conditions.

A marking of the net is a subset M of positions of P.[2] A marked net is a structure (N, Init), where N is a net and Init is an initial marking. A transition t is *enabled* at a marking M if ${}^{\bullet}t \subseteq M$. The transition relation is defined on the set of markings: $M \xrightarrow{t} M'$ if a transition t is enabled at in M and $M' = (M \setminus {}^{\bullet}t) \cup t^{\bullet}$. Let $\xrightarrow{*}$ denote the reflexive and transitive closure of $\longrightarrow$. A marking M' *covers* a marking M if $M \subseteq M'$.

A *firing sequence* is a finite or infinite sequence of transitions $t_1 t_2 \ldots$ provided we have a sequence of markings $M_0 M_1 \ldots$ such that $M_0 = \text{Init}$ and for each i, $M_i \xrightarrow{t_{i+1}} M_{i+1}$. We denote the set of firing sequences of (N, Init) as $FS(N, \text{Init})$. A firing sequence can be viewed as a sequential execution of the Petri net. However, we are interested in the partially-ordered runs that the Petri net exhibits; we will define these using Mazurkiewicz traces.

Traces: A *trace alphabet* is a pair (Σ, I) where Σ is a finite alphabet of actions and $I \subseteq \Sigma \times \Sigma$ is an irreflexive and symmetric relation over Σ called the *independence* relation. The induced relation $D = (\Sigma \times \Sigma) \setminus I$ (which is symmetric and reflexive) is called the *dependence* relation. A Mazurkiewicz trace is a behavior that describes a partially-ordered execution of events in Σ (when $I = \emptyset$, it is simply a word).

Definition 2. *[4] A (Mazurkiewicz)* trace *over the trace alphabet (Σ, I) is a Σ-labeled poset $t = (\mathcal{E}, \preceq, \lambda)$ where $\mathcal{E}$ is a finite or a countable set of events, $\preceq$ is a partial order on $\mathcal{E}$, called the* causal order*, and $\lambda : \mathcal{E} \longrightarrow \Sigma$ is a labeling function such that the following hold:*

[2] Petri nets can be more general, but in this paper we restrict to 1-safe Petri nets where each place gets at most one token.

– $\forall e \in \mathcal{E}, \downarrow e$ *is finite.* *Here,* $\downarrow e = \{e' \in E \mid e' \leq e\}$.
So we demand that there are only finitely many events causally before e.
– $\forall e, e' \in \mathcal{E}, e \prec\!\!\cdot\, e' \Rightarrow \lambda(e) D \lambda(e')$.[3] *Events that are immediately causally related must correspond to dependent actions.*
– $\forall e, e' \in \mathcal{E}, \lambda(e) D \lambda(e') \Rightarrow (e \preceq e' \ \vee e' \preceq e)$. *Any two events with dependent labels must be causally related.*

$\mathcal{T}(\Sigma, I)$ denotes the set of all traces over (Σ, I). We identify traces that are isomorphic.

A *linearization* of a trace $t = (\mathcal{E}, \preceq, \lambda)$ is a linearization of its events that respects the partial order; in other words, it is a word structure $(\mathcal{E}, \preceq', \lambda)$ where $\preceq'$ is a linear order with $\preceq \ \subseteq \ \preceq'$.

Let us define an equivalence on words over Σ: $\sigma \sim \sigma'$ if and only if for every pair of letters $a, b \in \Sigma$, with aDb, $\sigma \downarrow \{a, b\} = \sigma' \downarrow \{a, b\}$, where $\downarrow$ is the projection operator that drops all symbols not belonging to the second argument. Then, σ and σ' are linearizations of the same trace iff $\sigma \sim \sigma'$. We denote the equivalence class that σ belongs to as $[\sigma]$.

Let (Σ, I) be a trace alphabet and $\sim$ be the associated relation. Let us now formally associate the (unique) trace that corresponds to a word σ over Σ.

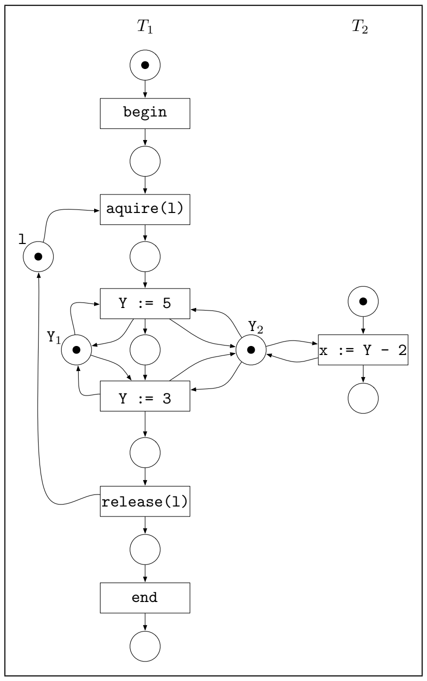


Fig. 3. Sample Net Model

A finite word σa is said to be *prime* if for every $\sigma' \sim \sigma a$, σ' is of the form $\sigma'' a$ (i.e. all words equivalent to σa end with a).

Let σ be a finite or infinite word over Σ. The trace associated with σ, $Tr(\sigma) = (\mathcal{E}, \preceq, \lambda)$ is defined as:

– $\mathcal{E} = \{[\sigma'] \mid \sigma'$ is prime $, \exists \sigma'' \sim \sigma, \sigma'$ is a prefix of $\sigma''\}$,
– $[\sigma] \preceq [\sigma']$ if there exists $\sigma_1 \in [\sigma], \sigma'_1 \in [\sigma']$ such that σ_1 is a prefix of σ'_1,
– $\lambda([\sigma' a]) = a$ for each $[\sigma' a] \in \mathcal{E}$.

It is easy to see that $Tr(\sigma)$ is a trace, and σ is a linearization of it.

Traces of a Petri net: Let us now define the set of traces generated by a Petri net. Given a marked net (N, Init), $N = (P, T, F)$, we consider the trace alphabet (Σ, I) where $\Sigma = T$, and $(t, t') \in I$ if and only if the neighborhoods of t and t' are disjoint, i.e. $({}^{\bullet}t \cup t^{\bullet}) \cap ({}^{\bullet}t' \cup t'^{\bullet}) = \emptyset$.

[3] $\prec\!\!\cdot$ is the immediate causal relation defined as: $e \prec\!\!\cdot\, e'$ iff $e \prec e'$ and there is no event e'' such that $e \prec e'' \prec e'$.

Now the traces generated by the net is is defined as $\{ Tr(\sigma) \mid \sigma \in FS(N, \text{Init})\}$. Note that a single trace represents several sequential runs, namely all its linearizations.

3 Causal Atomicity

Modeling programs using Petri Nets

We model the flow of control in SML programs by Petri nets. This modeling formally captures the concurrency between threads using the concurrency constructs of a Petri net, captures synchronizations between threads (e.g.. locks, accesses to global variables) using appropriate mechanisms in the net, and formalizes the fact that data is abstracted in a sound manner.

Figure 4 illustrates the function N that models statements using nets (inductively, for a fixed number of threads n). $N(\mathtt{S})$ is defined to have a unique *entry* place $p_{in}^{\mathtt{S}}$ and one or more *exit* transitions $tx_1^{\mathtt{S}}, \ldots, tx_m^{\mathtt{S}}$. In this natural way of modeling the control of a program, transitions correspond to program statements, and places are used to control the flow, and model the interdependencies and synchronization primitives. Figure 3 illustrates the Petri net model for the program in Figure 5(a).

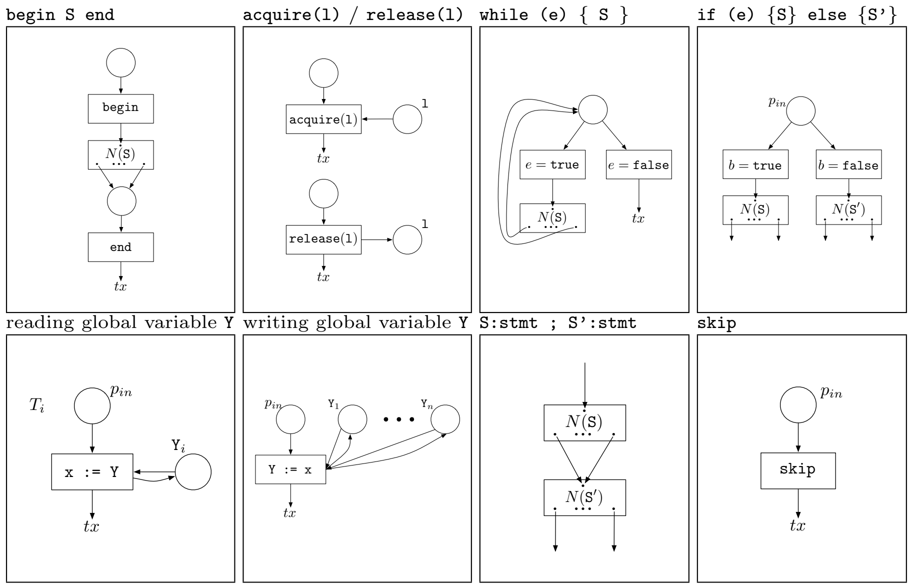


Fig. 4. Model Construction

There is a place l associated to each lock l which initially has a token in it. To acquire a lock, this token has to be available which then is taken and put back when the lock is released. This ensures that at most one thread can hold the lock at any time.

For each global variable $\mathtt{Y}$, there are n places $\mathtt{Y}_1, \ldots, \mathtt{Y}_n$, one per thread. Every time the thread T_i reads the variable $\mathtt{Y}$ ($\mathtt{Y}$ appears in an expression), it takes the token from the place $\mathtt{Y}_i$ and puts it back immediately. If T_i wants to write $\mathtt{Y}$ ($\mathtt{Y}$ is on the left side of an assignment), it has to take one token from each place $\mathtt{Y}_j$, $1 \leq j \leq n$ and put them all back. This is to ensure causality: two read operations of the same variable by different threads will be independent (as their neighborhoods will be disjoint), but a read and a write, or two writes are declared dependent. If $N_i = (P_i, T_i, F_i)$ is the Petri net model for statement $\mathtt{S}_i$ ($1 \leq i \leq n$), then the Petri net model for $\mathtt{S}_1 \mid\mid \cdots \mid\mid \mathtt{S}_\mathtt{n}$ is the net $(P_1 \cup \cdots \cup P_n, T_1 \cup \cdots \cup T_n, F_1 \cup \cdots \cup F_n)$, assuming T_i's are disjoint. Note the soundness of the abstraction: if a read and a write of two threads are simultaneously enabled (i.e. if there is a race condition), then the order on their accesses *may* be crucial. Since we are not keeping track of data in any manner, we declare them to be causally dependent and hence will consider the two runs inequivalent. The dependency relation defined in the model will lead to the appropriate notion of causality in the traces of the Petri net.

For a firing sequence σ of the net corresponding to a program, the sequence σ may not be feasible in the concrete program (because of the abstraction of data values). However, note that for every feasible sequence of the concrete program, its control trace is a trace of the net. Moreover, if σ is a firing sequence of the net which is feasible in the program (say by a concrete run r), then it is easy to see that for *each* firing sequence σ' such that $\sigma' \in [\sigma]$, there is a concrete run r' corresponding to it in the program that is equivalent to r (in terms of resulting in the same valuation of concrete variables). This property is key in ensuring that our entire approach is sound, as we will use trace equivalence as the equivalence over runs in defining atomicity.

Causal Atomicity

Recall the general notion of atomicity: a block is atomic if for for every sequential execution in which it is executed, there is another *equivalent* sequential execution where the block is executed without being interleaved with other threads. Given our abstraction using a Petri net, the only reasonable definition of equivalence of two sequential executions is that they are linearizations of the *same control trace* (see argument above).

Let us first illustrate the concept of causal atomicity by a simple example. Consider the two programs in Figure 5. Although the first thread (on the left) is the same in both versions, the block within `begin` and `end` is atomic in 5(b) and not atomic in 5(a).

The Petri net model of a program P induces the traces that correspond to the partially ordered runs of the program, which we call the *control traces*. *Causal atomicity* is defined as a property of these control traces. Figure 6 shows a control trace of the non-atomic program of Figure 5(a). Here labels of the events (transitions given by λ) are mentioned instead of the event names themselves

```
       T                    T'                         T                 T'

lock l ; int Y;      ||                   lock l ; int Y;      ||
(1) begin            (1) x := Y - 2       (1) begin            (1) acquire(l)
(2)   acquire(l)                          (2)   acquire(l)     (2) x := Y - 2
(3)     Y := 5;                           (3)     Y := 5;       (3) release(l)
(4)     Y := 3                            (4)     Y := 3
(5)   release(l)                          (5)   release(l)
(6) end                                   (6) end
                    (a)                                       (b)
```

Fig. 5. Two Programs

to make the trace more readable. The arrows depict the immediate causality relation. The trace is a witness for non atomicity since `x:=Y-2` is causally after `Y:=5` and causally before `Y:=3`, and therefore in all linearizations of this trace, `x:=Y-2` has to appear in the middle of the block.

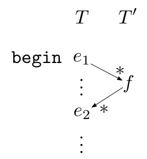
$$\begin{tikzcd}
\texttt{begin} \arrow[r] & \texttt{acquire(l)} \arrow[r] & \mathtt{Y := 5} \arrow[dr] & & \mathtt{Y := 3} \arrow[r] & \texttt{release(l)} \arrow[r] & \texttt{end} \\
& & & \mathtt{x := Y - 2} \arrow[ur] & & &
\end{tikzcd}$$

Fig. 6. Non-Atomic Trace

A notational remark: when we denote a transition as t^{T_i}, we mean that it belongs to the thread T_i.

Definition 3. *A code block $B =$ `begin S end` of the program P is **causally atomic** if and only if the Petri net model of the program P does not induce a trace $Tr = (mathcalE, \preceq, \lambda)$ for which the following holds:*

$\exists e_1, e_2, f \in \mathcal{E} : \; e_1 \preceq f \preceq e_2$ *where*
$\lambda(e_1) = t^T_{\texttt{begin}}, \lambda(e_2) = t^T_2, \lambda(f) = t^{T'}_3$ *such that*
$T \neq T'$ *and* $\not\exists e \in \mathcal{E} : (\lambda(e) = t^T_{\textsf{end}} \;\wedge\; e_1 \preceq e \preceq e_2)$



The above definition says that a block declared atomic is not causally atomic if the block begins, and there are two events, e_2 belonging to the same thread (and e_2 occurs before the block finishes) and f belonging to another thread such that f occurs causally between the beginning of the block and e_2. Note that traces that witness non-atomicity may not even execute the block completely (and we do not require any termination requirement for blocks).

The following theorem captures the intuition of why the above defines causal atomicity; it argues that if a trace of the program is not of the kind mentioned in the definition above, then there is indeed some linearization of events that executes the atomic block without interleaving. The proof is easy for finite traces, but more involved for the infinite ones; we skip the proofs.

Theorem 1. (a) *A code block $B =$ `begin S end` of the program P is causally atomic if and only if for all finite traces induced by the Petri net model of P, there is a linearization of the trace where all occurrences of block B occur sequentially (without interleaving with other threads).*

(b) *If a code block $B =$ `begin S end` of thread T in the program P is causally atomic then for all infinite traces induced by the Petri net model of P, there is a linearization of a causally downward closed subset of the events of the trace that contains all events belonging thread T, in which occurrences of block B occur sequentially.*

Note that the above theorems yield soundness of our approach: if a code block B is causally atomic, then by the above theorem and by the fact that either every linearization of a trace of the net is feasible in the concrete program or none are, it follows that the block B is atomic in the concrete program as well.

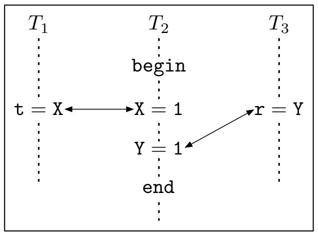


Fig. 7. Example

The program in Figure 7 distinguishes causal atomicity from other static notions of atomicity in the literature. The code block in thread T_2 is causally atomic. However, since there are races on both global variables `X` and `Y`, both statements `X = 1` and `Y = 1` are *non-mover* statements and this block is *not a transaction*, and therefore will not be detected as atomic by the method in [10]. Our notion of causal atomicity is also *behavioral* and more geared towards model checking as it depends on the partial-order executions of the program, not on the static structure of the code.

Commit-atomicity [7] is a dynamic notion of atomicity which is different from our static notion. The presence of data in commit-atomicity allows a more precise detection of atomicity according to the general definition of atomicity and there are examples (e.g. see Bluetooth3 in Section 5) that can be detected atomic by commit-atomicity, but they fail the causal atomicity check. On the other hand, the presence of data limits commit-atomicity to finite state space programs, and impedes scalability (specifically, in terms of number of threads). However, causal atomicity can deal with infinite data since the data is completely abstracted. Also, the commit-atomicity method requires the the atomic blocks to be terminating while we do not need such an assumption.

4 Checking Atomicity

In this section, we present how causal atomicity defined on traces can be reduced to coverability in a colored Petri net.

Colored Petri Nets

Colored Petri nets are subclass of *high level Petri nets* [14]. We explain how causal atomicity checks can be done by checking very simple properties on the colored nets. This does not imply any complications theoretically since the result in [14] shows that each colored net has an equivalent Petri net, and practically since most Petri net analysis tools today support high level nets.

We use a very simplified definition of colored Petri nets. We will not define them formally, but explain them intuitively. A colored Petri net has, as an underlying structure, a Petri net, but a token at a place can take a color c, which is drawn from a finite set of colors C. Transitions get enabled when their pre-conditions get marked, as before; but the transitions can now examine the colors of tokens and decide the colors of tokens on the post-conditions.

Given a model of a program P as a net $N = (P, T, F)$ and an initial marking Init, we define a colored Petri net that has the same underlying net N, but with colors $C = \{A, B, Y, R\}$. The initial marking is the initial marking Init, with all tokens colored A (achromatic).

The evolution of colors is determined by rules defined below. Note however that since the colored net has the same underlying net, it inherits the independence relation and generates the same traces as the net modeling the program.

Tokens are of one of the colors achromatic (A), blue (B), yellow (Y), and red (R), and we use them to monitor executions. The net nondeterministically chooses an occurrence of a block B of a thread T that is to be checked for atomicity, and at its `begin` event, turns the local control place of T to the color blue. Whenever an event has a local pre-condition marked blue, it taints all its post-conditions blue; hence the blue color spreads to all conditions causally after the `begin`-event of B. When a different thread reads a blue token, it transforms the token to yellow. Events that read any yellow-colored pre-condition taint their post-conditions to yellow as well, and hence propagate the yellow color to causal successors. If the thread T executes a statement of block B (before reaching the end of the block) and reads a pre-condition labeled yellow, it would detect violation of causal atomicity, and mark a special place red. If the end of block B is reached without detecting violation of causal atomicity, the net abruptly stops, as the occurrence of the block guessed to be non causal atomic was wrong.

Thus the problem of checking atomicity in the Petri net model of a program reduces to the problem of checking whether in the associated colored net, there is a reachable marking that covers a special place colored red (R).

Theorem 2. *The special place with a red (R) token is coverable in the colored Petri net constructed from the Petri net model of the program if and only if some marked block B is not causally atomic.*

5 Experiments

We have applied the method in Section 4 to check causal atomicity of several programs taken from [7]. This section presents a brief description of each program and the performance of our algorithm.

Dekker's mutual exclusion algorithm: Dekker's algorithm is a classic algorithm for mutual exclusion between two threads that uses subtle synchronization. The mutual exclusion is modeled by means of two boolean variables. We check whether the critical sections of the threads are atomic, and they do turn out to be causally atomic.

Busy-waiting acquire lock: In this example a busy waiting while loop is used to acquire a lock. There is a forever loop that acquires the mutex using this method, then a global variable `data` is updated, and the mutex is released. The correctness specification requires the updating of the data to be done atomically. We have checked two different versions of this example. In Acquire1 there is only one line modifying the data, while in Acquire2 there are several lines manipulating the data. Using our technique, there is negligible difference difference in the size of the unfolding between the two cases since the partial order semantics would not interleave the internal statements that modify the data. In contrast, the model checking algorithm in [7] uses interleavings, and they see a large increase in the time taken for Acquire2. One can make the block in Acquire1 non-atomic by adding an extra thread that manipulates the data without acquiring the mutex; nAcquire1 refers to this case.

In Acquire1* and Dekker*, multiple blocks (one per thread) are checked for causal atomicity. This is in contrast to the rest of the benchmarks where one block is checked at a time.

Bluetooth Device Driver: We used a simplified version of the Bluetooth device driver in Windows NT (Bluetooth), similar to the one used in [20, 7]. There are two dispatch functions; let us call them `Add` and `Stop`. Any process that wants to use the driver calls `Add`, and any process that wants to stop the driver calls `Stop`. The correctness specification requires these two dispatch functions to run atomically. The `Add` function is not causally atomic which can be verified using only two threads where one calls `Add` and the other one calls `Stop`. This turns out to be a real cause for concern in the code, as interleaving events from other threads while executing `Add` does make the program behave unexpectedly; this was already reported in [20, 7]. There is a fixed version of Bluetooth from [3] (Bluetooth3) which is still not causally atomic despite the fact that it is correct. However, *commit-atomicity* [7] can detect this as atomic since it can keep track of the value of the counter in the program.

Experimental Results

Table 1 shows the result of evaluating the above benchmarks using Pep [12]. Each program is modeled as a colored Petri net as described in Section 4. The unfolding of the colored net is generated. Then, with a simple query, we check whether the special place having a red token is coverable. The table reports the size of the unfolding, the time taken to check for causal atomicity (in seconds), and whether the atomicity checker detected causal atomicity. We performed these experiments under Linux on a 1.7GHz Pentium M laptop with 384MB of memory. The output time is reported with the precision of 10 milliseconds, and all experiments with 0 reported time were done in less than 10ms.

Note that in the Acquire1 example, the size of the unfolding grows only linearly with the number of threads; this reflects the space savings obtained through unfoldings. In contrast, the model checking algorithm in [7], which reasons using sequential traces, started to fail at around four threads. Note however that this isn't a fair comparison as the notion of atomicity (called *commit-atomicity*) in

Benchmark	Causally Atomic?	#Threads	Unfolding #Places	Unfolding #Events	Time (sec)
Dekker	Yes	2	52	24	0
Dekker*	Yes	2	795	409	0.01
Acquire1	Yes	4	81	34	0
Acquire1	Yes	30	2135	1022	0.03
Acquire1	Yes	100	21105	10402	3.71
Acquire1	Yes	150	46655	23102	22.31
Acquire1*	Yes	4	146	56	0
Acquire1*	Yes	30	4202	1200	0.35
Acquire1*	Yes	50	10582	2985	5.02
Acquire1*	Yes	100	40486	10784	635.56
nAcquire1	No	4	97	43	0
nAcquire1	No	6	171	77	0
nAcquire1	No	8	261	119	0
Acquire2	Yes	4	73	30	0
Acquire2	Yes	6	146	74	0
Bluetooth	No	2	235	116	0
Bluetooth3	No	2	223	109	0

Table 1. Programs and Performances

[7] is quite different, more accurate, and harder to check. However, in all the examples except Bluetooth3, their notion of atomicity agreed with ours. All the experiments can be found at: `http://peepal.cs.uiuc.edu/~azadeh/atomicity`.

6 Conclusions

We have defined a notion of atomicity based on the causal structure of events that occur in executions of programs. The causal structure is obtained using a straightforward data abstraction of the program that captures control interactions between threads using the concurrent model of Petri nets. We have demonstrated the usefulness of the notion of causal atomicity and shown that it can be effectively checked using unfolding based algorithms on Petri nets.

This work is part of a bigger project whose aim is to identify sound control abstractions for concurrent programs that can be used for static analysis (for example, dataflow analysis). We believe that true concurrent models (such as Petri nets) and true concurrent behaviors (like traces and event structures) would prove to be effective for this purpose. This paper demonstrates the efficacy of a truly concurrent behavior model (traces) in identifying atomicity.

There are several future directions. Our method of checking atomicity is a *global analysis* involving all threads simultaneously, while methods based on types and transactions work *locally* on each thread independently. Since local algorithms are likely to scale better, it would be interesting to find the weakest local property that ensures global causal atomicity. Also, finding *compositional* algorithms that derive information from each program locally and combine these to check for global atomicity would be interesting to study as they would scale better than global analysis and be more accurate than local analysis. Finally, we would also like to study extensions of atomicity defined in the literature (for example, *purity* [9]), in the causal setting.

References

1. R. Alur, K. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000.
2. P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman., 1987.
3. S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *TACAS*, volume 3920 of *LNCS*, pages 334–349, 2006.
4. V. Diekert and G. Rozenberg. *The Book of Traces*. World Scientific Publishing Co., 1995.
5. J. Esparza, S. Romer, and W. Vogler. An improvement of McMillan's unfolding algorithm. In *TACAS*, volume 1055 of *LNCS*, pages 87 – 106, 1996.
6. K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.
7. C. Flanagan. Verifying commit-atomicity using model-checking. In *SPIN*, pages 252–266, 2004.
8. C. Flanagan and S. Freund. Atomizer: a dynamic atomicity checker for multi-threaded programs. In *POPL*, pages 256–267, 2004.
9. C. Flanagan, S. Freund, and S. Qadeer. Exploiting purity for atomicity. *IEEE Trans. Software Eng.*, 31(4):275–291, 2005.
10. C. Flanagan and S. Qadeer. Types for atomicity. In *TLDI*, pages 1 – 12, 2003.
11. M. Flé and G. Roucairol. On serializability of iterated transactions. In *PODC*, pages 194–200, 1982.
12. B. Grahlmann. The PEP tool. In *CAV*, pages 440–443, 1997.
13. J. Hatcliff, Robby, and M. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *VMCAI*, pages 175–190, 2004.
14. K. Jensen. *Coloured Petri nets (2nd ed.): basic concepts, analysis methods and practical use: volume 1*. Springer-Verlag, London, UK, 1996.
15. R. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
16. K. Lodaya, M. Mukund, R. Ramanujam, and P. S. Thiagarajan. Models and logics for true concurrency. Technical Report TCS–90–3, School of Mathematics Internal, 1990.
17. K. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.
18. M. Nielsen, G. Plotkin, and G. Winsker. Peri nets, event structures and domains — part i. *Theoretical Computer Science*, 13:85 – 108, 1981.
19. C. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, Inc., New York, NY, USA, 1986.
20. S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.
21. A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, 5th edition, 2005.
22. L. Wang and S. Stoller. Run-time analysis for atomicity. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.
23. L. Wang and S. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *PPOPP*, pages 61–71, 2005.