Causal Dataflow Analysis for Concurrent Programs

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

Abstract. We define a novel formulation of dataflow analysis for concurrent programs, where the flow of facts is along the *causal* dependencies of events. We capture the control flow of concurrent programs using a Petri net (called the *control net*), develop algorithms based on partially-ordered unfoldings, and report experimental results for solving causal dataflow analysis problems. For the subclass of distributive problems, we prove that complexity of checking data flow is linear in the number of facts and in the *unfolding* of the control net.

1 Introduction

Advances in multicore technology and the wide use of languages that inherently support threads, such as Java, foretell a future where concurrency will be the norm. Despite their growing importance, little progress has been made in static analysis of concurrent programs. For instance, there is no standard notion of a control-flow graph for concurrent programs, while the analogous notion in sequential programs has existed for a long time [1]. Consequently, dataflow analysis problems (arguably the simplest of analysis problems) have not been clearly understood for programs with concurrency.

While it is certainly easy to formulate dataflow analysis for concurrent programs using the *global product state space* of the individual threads, the usefulness of doing so is questionable as algorithms working on the global state space will not scale. Consequently, the literature in flow analysis for threaded programs concentrates on finding tractable problem definitions for dataflow analysis. A common approach has been to consider programs where the causal relation between events is *static* and apparent from the structure of the code (such as fork-join formalisms), making feasible an analysis that works by finding fixpoints on the *union* of the individual sequential control flow graphs. These approaches are often highly restrictive (for example, they require programs to have no loops [2] or at least to have no loops with concurrent fork-join constructs [3, 4]), and cannot model even simple shared-memory program models. In fact, a coherent formulation of control-flow that can capture programs with dynamic concurrency (including those with shared memory) and a *general definition* of dataflow analysis problems for these programs has not been formulated in the literature (see the end of this section for details on related work).

The goals of this paper are (a) to develop a formal *control-flow* model for programs using Petri nets, (b) to propose a novel definition of dataflow analyses based on *causal flows* in a program, (c) to develop algorithms for solving causal flow analyses when the domain of flow facts is a finite set $\mathbb{D}$ by exploring the partially-ordered runs of the program as opposed to its interleaved executions, and (d) to provide provably efficient algorithms for the class of *distributive* CCD problems, and support the claim with demonstrative experiments. The framework we set forth in this paper is the first one we know that defines a formal general definition of dataflow analysis for concurrent programs.

We first develop a Petri net model that captures the control flow in a concurrent program, and give a translation from programs to Petri nets that explicitly abstracts data and captures the control flow in the program. These nets, called *control nets*, support dynamic concurrency, and can model concurrent constructs such as lock-synchronizations and shared variable accesses. In fact, we have recently used the same model of control nets to model and check *atomicity* of code blocks in concurrent programs [5]. We believe that the control net model is an excellent candidate for capturing control flow in concurrent programs, and can emerge as the robust analog of *control-flow graphs* for sequential programs.

The causal concurrent dataflow (CCD) framework is in the flavor of a *meet-over-all-paths* formulation for sequential programs. We assume a set of dataflow facts $\mathbb{D}$ and each statement of the program is associated with a *flow transformer* that changes a subset of facts, killing some old facts and generating new facts. However, we demand that the flow transformers respect the concurrency in the program: we require that if two *independent* (concurrent) statements transform two subsets of facts, D and D', then the sets D and D' must be *disjoint*. For instance, if there are two local variable accesses in two different threads, these statements are independent, and cannot change the same dataflow fact, which is a very natural restriction. For example, if we are tracking *uninitialized variables*, two assignments in two threads to local variables do affect the facts pertaining to these variables, but do not modify the same fact. We present formulations of most of the common dataflow analysis problems in our setting.

The structural restriction of requiring transformers to respect causality ensures that dataflow facts can be inferred using partially ordered traces of the control net. We define the dataflow analysis problem as *a meet over partially ordered traces* that reach a node, rather than the traditional meet-over-paths definition. The meet-over-traces definition is crucial as it preserves the concurrency in the program, allowing us to exploit it to solve flow analysis using *partial-order* based methods, which do not explore all interleavings of the program.

Our next step is to give a solution for the general causal dataflow analysis problem when the set of of facts $\mathbb{D}$ is finite by reducing the problem to a reachability problem of a Petri net, akin to the classic approach of reducing meet-over-paths

to graph reachability for sequential recursive programs [6]. Finally, the reachability/coverability problem is solved using the optimized partial-order *unfolding* [7, 8] based tool called PEP [9].

For the important subclass of *distributive* dataflow analysis problems, we develop a more efficient algorithm for checking flows. If N is the control net of a program and the size of its finite unfolding is n, we show that any distributive CCD problem over a domain $\mathbb{D}$ of facts results in an augmented net of size $n|\mathbb{D}|$ (and hence in an algorithm working within similar bounds of time and space). This is a very satisfactory result, since it proves that the causal definition does not destroy the concurrency in the net (as that would result in a blow-up in n), and that we are exploiting distributivity effectively (as we have a linear dependence on $|\mathbb{D}|$). The analogous result for sequential recursive programs also creates an augmented graph of size $n|\mathbb{D}|$, where n is the size of the control-flow graph.

Related Work. Although the majority of flow analysis research has focused on sequential software [10–13], flow analysis for concurrent software has also been studied to some extent. Existing methods for flow-sensitive analyses have at least one of the following restrictions: (a) the programs handled have simple static concurrency and can be handled precisely using the union of control flow graphs of individual programs, or (b) the analysis is sound but not complete, and solves the dataflow problem using heuristic approximations.

A body of work on flow-sensitive analyses exists in which the model for the program is essentially a collection of CFGs of individual threads (tasks, or components) together with additional edges among the CFGs that model inter-thread synchronization and communication [14–16]. These analyses are usually restricted to a class of behaviors (such as detecting deadlocks) and their models do not require considering the set of interleavings of the program. More general analyses based on the above type of model include [17] which presents a *unidirectional* bit-vector dataflow analysis framework based on abstract interpretation (where the domain $\mathbb{D}$ is a *singleton*). This framework comes closest to ours in that it explicitly defines a meet-over-paths definition of dataflow analysis, can express a variety of dataflow analysis problems, and gives sound and complete algorithms for solving them. However, it cannot handle dynamic synchronization mechanisms (such as locks), and the restriction to having only one dataflow fact is crucially (and cleverly) used, making multidimensional analysis impossible. For example, this framework cannot handle the problem of solving *uninitialized variables*. See also [2] for dataflow analysis that uses flow along causal edges but disallows loops in programs and requires them to have static concurrency. The works in [3, 4] use the extension of the static single assignment form [18] for concurrent programs with emphasis on optimizing concurrent programs as opposed to analyzing them.

In [19], concurrent models are used to represent interleavings of programs, but the initial model is coarse and refined to obtain precision, and efficiency is gained by sacrificing precision. Petri nets are used as control models for Ada programs in [20], although the modeling is completely different form ours. In [21], the authors combine reachability analysis with symbolic execution to prune the infeasible paths in order to achieve more effective results.

2 Preliminaries

A Simple Multithreaded Language. We base our formal development on the language SML (Simple Multithreaded Language). Figure 1 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.

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

Fig. 1. SML syntax

Petri Nets and Traces:

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.[1] A marked net is a structure (N, M_0),

[1] 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.

where N is a net and M_0 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 transition t is enabled at 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 sequence of transitions $t_1 t_2 \ldots$ provided we have a sequence of markings $M_0 M_1 \ldots$ and for each i, $M_i \xrightarrow{t_{i+1}} M_{i+1}$. We denote the set of firing sequences of (N, M_0) as $FS(N, M_0)$. Given a marked net (N, M_0), $N = (P, T, F)$, the *independence* relation of the net I_N is defined as $(t, t') \in I$ if the neighborhoods of t and t' are disjoint, i.e. $({}^\bullet t \cup t^\bullet) \cap ({}^\bullet t' \cup t'^\bullet) = \emptyset$. The *dependence* relation D_N is defined as the complement of I_N.

Definition 1. *A* trace *of a marked net (N, M_0) is a labeled poset $Tr = (\mathcal{E}, \preceq, \lambda)$ 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 $\lambda : \mathcal{E} \longrightarrow T$ is a labeling function such that the following hold:*

- $\forall e, e' \in \mathcal{E}, e \prec\!\!\!\cdot\; e' \Rightarrow \lambda(e) D_N \lambda(e')$.[2] *Events that are immediately causally related must correspond to dependent transitions.*
- $\forall e, e' \in \mathcal{E}, \lambda(e) D_N \lambda(e') \Rightarrow (e \preceq e' \ \vee e' \preceq e)$. *Any two events with dependent labels must be causally related.*
- *If σ is a linearization of Tr then $\sigma \in FS(N, M_0)$.*

For any event e in a trace $(\mathcal{E}, \preceq, \lambda)$, define $\downarrow e = \{e' \in \mathcal{E} \mid e' \preceq e\}$ and let $\Downarrow e = \downarrow e \setminus \{e\}$.

Petri Nets and Traces: We briefly define nets and traces, and refer a reader unfamiliar with these concepts to Appendix **??**.

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.[3] 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 transition t is enabled at 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,

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

[3] 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.

$M_i \xrightarrow{t_{i+1}} M_{i+1}$. 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. *[22] 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:*

- $\forall e \in \mathcal{E}$, $\downarrow e$ *is finite.* *Here,* $\downarrow e = \{e' \in E \mid e' \preceq e\}$.
  So we demand that there are only finitely many events causally before e.
- $\forall e, e' \in \mathcal{E}, e \prec\!\!\!\cdot\ e' \Rightarrow \lambda(e) D \lambda(e')$.[4] *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.

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).

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

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

- $\mathcal{E} = \{[\sigma'] \mid \sigma' \text{ is prime }, \exists \sigma'' \sim \sigma, \sigma' \text{ is a prefix of } \sigma''\}$,
- $[\sigma] \preceq [\sigma']$ 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. $({}^\bullet t \cup t^\bullet) \cap ({}^\bullet t' \cup t'^\bullet) = \emptyset$.

Now the traces generated by the net is is defined as $\mathbb{T}_{\text{Init}}(N) = \{\, Tr(\sigma) \mid \sigma \in FS(N, \text{Init})\}$. Note that a single trace represents several sequential runs, namely all its linearizations. We will omit N and Init wherever they can be inferred from the context.

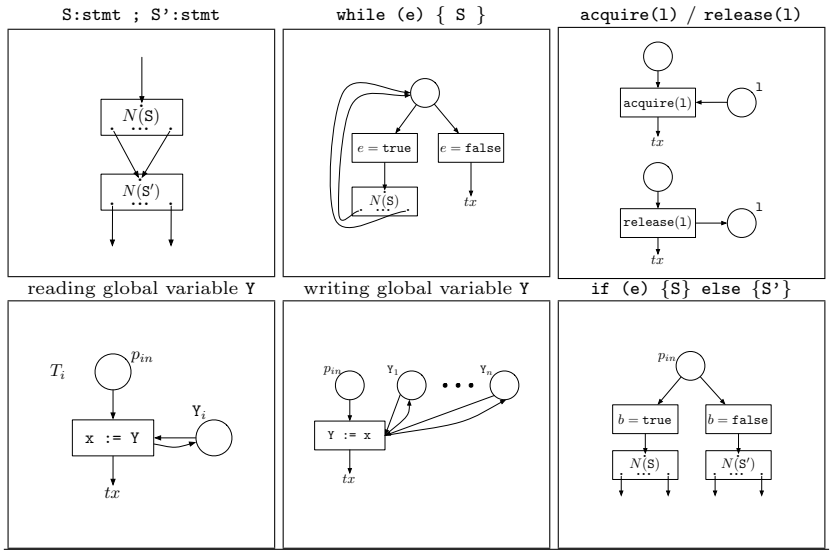


Fig. 2. Control Net Construction

3 The Control Net of a Program

We model the flow of control in SML programs using Petri nets. We call this model the *control net* of the program. The control net 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 Petri net, and formalizes the fact that data is abstracted in a sound manner.

We describe the main ideas of this construction but skip the details (see [23] for details). Transitions in the control net correspond to program statements, and places are used to control the flow, and to model the interdependencies and synchronization primitives. Figure 3 illustrates a program and its control net.

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.

For each global variable `Y`, there are n places $\mathtt{Y}_1, \ldots, \mathtt{Y}_n$, one per thread. Every time the thread T_i reads the variable `Y` (`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 `Y` (`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 ensures correct 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 to a variable are declared dependent.

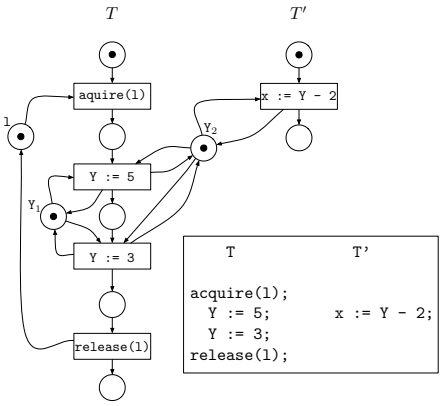


Fig. 3. Sample Net Model

4 Causal Concurrent Dataflow Framework

We now formulate our framework for dataflow analysis of concurrent programs based on causality, called the Causal Concurrent Dataflow (CCD) framework.

A property space is a *subset lattice* $(\mathcal{P}(\mathbb{D}), \sqsubseteq, \sqcup, \bot)$ where $\mathbb{D}$ is a finite set of *dataflow facts*, $\bot \subseteq \mathbb{D}$, and where $\sqcup$ and $\sqsubseteq$ can respectively be $\cup$ and $\subseteq$, or $\cap$ and $\supseteq$. Intuitively, $\mathbb{D}$ is the set of dataflow facts of interest, $\bot$ is the initial set of facts, and $\sqcup$ is the *meet* operation that will determine how we combine dataflow facts along different paths reaching the same control point in a program. "May" analysis is formulated using $\sqcup = \cup$, while "must" analysis uses the $\sqcup = \cap$ formulation. The property space of an IFDS (interprocedural finite distributive subset) problem [6] for a sequential program (i.e. the subset lattice) is exactly the same lattice as above.

For every transition t of the control net, we associate two subsets of $\mathbb{D}$, D_t and D_t^*. Intuitively, D_t^* is the set of dataflow facts relevant at t, while $D_t \subseteq D_t^*$ is the subset of relevant facts that t may modify when it executes. The *transformation function* associated with t, f_t, maps every subset of D_t to a subset of D_t, reflecting how the dataflow facts change when t is executed.

Definition 3. *A causal concurrent dataflow (CCD) problem is a tuple* $(N, \mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{D}^*)$ *where:*

- $N = (P, T, F)$ *is the control net model of a concurrent program,*
- $\mathcal{S} = (\mathcal{P}(\mathbb{D}), \sqsubset, \sqcup, \bot)$ *is a property space,*
- $\mathcal{D} = \{D_t\}_{t \in T}$ *and* $\mathcal{D}^* = \{D_t^*\}_{t \in T}$*, where each* $D_t \subseteq D_t^* \subseteq \mathbb{D}$*.*
- $\mathcal{F}$ *is a set of functions* $\{f_t\}_{t \in T} : 2^{D_t} \to 2^{D_t}$ *such that:*
  (*) $\forall t, t' : (t, t') \in I_N \Rightarrow (D_t \cap D_{t'}^* = D_t^* \cap D_{t'} = \emptyset)$.[5]

We call a CCD problem *distributive* if all transformation functions in $\mathcal{F}$ are distributive, that is $\forall f_t \in \mathcal{F},\ \forall X, Y \subseteq D_t : f_t(X \sqcup Y) = f_t(X) \sqcup f_t(Y)$.

Remark 1. Condition (*) above is to be specially noted. It demands that for any two concurrent events e and e', e cannot change a dataflow fact that is relevant to e'. Note that if e and e' are events in a trace such that $D_{\lambda(e)} \cap D^*_{\lambda(e')}$ is non-empty, then they will be causally related.

4.1 Meet Over All Traces Solution

In a sequential run of a program, every event t has at most one predecessor t'. Therefore, the set of dataflow facts that hold before the execution of t (let us call this $in(t)$) is exactly the set of dataflow facts that hold after the execution of t' ($out(t')$). This is not the case for a trace (a partially ordered run). Consider the example in Figure 4. Assume t_1 generates facts d_1 and d_2, t_2 generates d_3 and t_3 kills d_2 and generates d_4. The corresponding D_t sets appear in the Figure. Trying to evaluate the "*in*" set of t_4, we see three important scenarios: (1) t_4 inherits *independent* facts d_3 and d_4 respectively from its immediate predecessors t_2 and t_3, (2) t_4 inherits fact d_1 from t_1 which is not its immediate predecessor, and (3) t_4 does not inherit d_2 from t_1 because t_3, which is a (causally) later event and the last event to modify d_2, kills d_2.

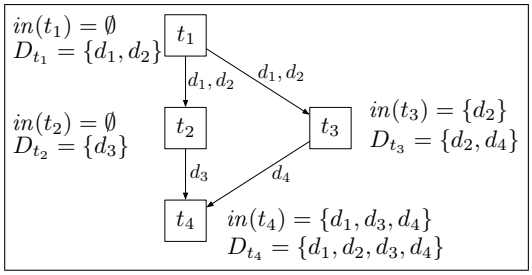


Fig. 4. Flow of facts over a trace.

This example demonstrates that in a trace the immediate causal predecessors do not specify the "*in*" set of an event. The indicating event is actually the

[5] And hence $D_t \cap D_{t'} = \emptyset$.

(causally) last event that can change a dataflow fact (eg. t_3 for fact d_2 in computing $in(t_4)$). We formalize this concept by defining the operator $maxc^d_{\preceq}(Tr)$, for a trace $Tr = (E, \preceq, \lambda)$ as $maxc^d_{\preceq}(Tr) = max_{\preceq}(\{e \mid e \in E \wedge d \in D_{\lambda(e)}\})$. Note that this function is undefined on the empty set, but well-defined on non-empty sets because all events that affect a dataflow fact d are causally related due to (*) in Definition 3.

Remark 1 suggests that for each event e it suffices to only look at the facts that are in the "*out*" set of events in $\Downarrow e$ (events that are causally before e), since events that are concurrent with e will not change any fact that's relevant to e.

Definition 4. *For any trace $Tr = (E, \preceq, \lambda)$ of the control net and for each event $e \in E$, we define the following dataflow sets:*

$$\begin{cases} in^{Tr}(e) = \bigcup_{d \in D^*_{\lambda(e)}} (out^{Tr}(maxc^d_{\preceq}(\Downarrow e)) \cap \{d\})) \\ out^{Tr}(e) = f_{\lambda(e)}(in^{Tr}(e) \cap D_{\lambda(e)}) \end{cases}$$

where $in^{Tr}(e)$ (respectively $out^{Tr}(e)$) indicates the set of dataflow facts that hold before (respectively after) the execution of event e of trace Tr.

In the above definition, $maxc^{d_i}_{\preceq}(\Downarrow e))$ may be undefined (if $\Downarrow e = \emptyset$), in which case we assume $in^{Tr}(e)$ evaluates to the empty set.

We can now define the **meet over all traces** solution for a program Pr, assuming the $\mathcal{T}(N)$ denotes the set of all traces induced by the control net N.

Definition 5. *The set of dataflow facts that hold before the execution of a transition t of a control net N is $MOT(t) = \bigcup_{Tr \in \mathcal{T}(N), e \in Tr, \lambda(e)=t} in^{Tr}(e)$.*

The above formulation is the concurrent analog of the meet-over-all-paths formulation for sequential programs. Instead of the above definition, we could formulate the problem as a meet-over-all-paths problem, where we take the meet over facts accumulated along the *sequential runs* (interleavings) of the concurrent program. However, due to the restriction (*) in Definition 3, we can show that the dataflow facts accumulated at an event of a trace is precisely the same as that accumulated using any of its linearizations. Consequently, for dataflow problems that respect causality by satisfying the condition (*), the meet-over-all-paths and the meet-over-traces formulations coincide. The latter formulation however yields faster algorithms based on partial-order methods based on unfoldings to solve the dataflow analysis problem.

4.2 The Global Meet Over All Paths Solution

A run of the program σ is a linearization of some trace depicted by the control net of the program. For each event e appearing in σ, let $\text{pre}_\sigma(e)$ (respectively

$\text{post}_\sigma(e)$) be the event happening immediately before (respectively after) e in σ. For the first event e_0 of each run, we assume $\text{pre}(e_0) = \perp$ and we also assume that $out^\sigma(\perp) = \perp$.

Definition 6. *For each run σ of the program and for each event e appearing in σ, we define the following two sets of dataflow facts:*

$$\begin{cases} in^\sigma(e) & = out^\sigma(pre_\sigma(e)) \\ out^\sigma(e) & = f_{\lambda(e)}(in^\sigma(e)) \end{cases}$$

where in^σ (respectively out^σ) is the set of dataflow facts which hold before (respectively after) execution of event e in the run σ.

Let $\mathbb{R}$ denote the set of all runs of the program Based on the definition of dataflow sets for a single run, we can define the *meet over all paths solution*:

Definition 7. *The set of all dataflow facts that hold before the execution of a transition t of a Petri net is $MOP(t) = \bigsqcup_{\sigma\in\mathbb{R},e\in\sigma,\lambda(e)=t} in^\sigma(e)|_{D_t^*}$ where we have $in^\sigma(e)|_{D_t^*} = in^\sigma(e) \cap D_t^*$.*

The restriction of the set $in^\sigma(e)$ to D_t^* ensures that only information relevant to an event can reach the event. If statement s of thread T is scheduled right after statement s' of thread T' in a run, but T and T' do not interact in any way, information from s' should not reach s merely because $s' = \text{pre}(s)$ in that particular run.

The Relation between MOP and MOT solutions

The following theorem shows that the two definitions MOP and MOT are equivalent.

Theorem 1. *For every CCD problem, we have $\forall t \in T, MOT(t) = MOP(t)$.*

Proof of Theorem 1:

Lemma 1. *For every trace Tr of the control net of a program, and for every event $e \in Tr$, and for every linearization σ of the trace Tr, $in^{Tr}(e) = in^\sigma(e)|_{D^*_{\lambda(e)}}$.*

Proof. (sketch)

(a) $d \in in^{Tr}(e) \Rightarrow d \in in^{\sigma}(e)|D^*_{\lambda(e)}$.
Since $in^{Tr}(e) \subseteq D^*_{\lambda(e)}$ by Definition 4, it suffices to show that $in^{Tr}(e) \subseteq in^{\sigma}(e)$, for all σ. We prove this by induction using the relation $\trianglelefteq$. The base case clearly holds for an empty trace. Assume that for trace Tr' such that $Tr' \trianglelefteq Tr$, the above statement holds. We prove then that it holds for Tr.
$d \in in^{Tr}(e)$ implies (by Definition 4) that there exists an event $e' = max^d_{\preceq}(\Downarrow e)$ such that $d \in out^{Tr}(e')$.
Consider any linearization σ of Tr. Consider the prefix of σ ending in e', and call it σ'. Consider the corresponding trace of σ', $Tr(\sigma')$. Clearly, $Tr(\sigma') \trianglelefteq Tr$. Therefore, by induction and the fact that $d \in out^{Tr}(e')$, we have $d \in out^{\sigma'}(e')$, or equivalently (since σ' is a prefix of σ) $d \in out^{\sigma}(e')$. We argue that there are no events in $\sigma \backslash \sigma'$ that can cancel d. Assume there is such an event e''. Since e'' changes d it has to be causally related to e'. Since e'' appears later than e' in σ, the only option is that $e' \preceq e''$. But this is in contradiction with $e' = max^d_{\preceq}(\downarrow e)$, therefore such event e'' cannot exist. If no event in $\sigma \backslash \sigma'$ cancels d, then d propagates to e and therefore we have $d \in in^{\sigma}(e)$.

(b) $d \in in^{\sigma}(e)|_{D^*_{\lambda(e)}} \Rightarrow d \in in^{Tr(\sigma)}(e)$.
We prove this by induction on length of σ. The base case clearly holds for the empty string and the corresponding empty trace. Assume $d \in in^{\sigma}(e)|_{D^*_{\lambda(e)}}$ for some linearization σ (ending in e) of trace Tr. This means that $d \in D^*_{\lambda(e)}$ and also, $d \in out^{\sigma}(pre(e))$. There are two possibilities:
   (i) $pre(e)$ generates d, and therefore $d \in D_{\lambda(pre(e))}$. By assumption (*) in Definition 3, $\lambda(pre(e))$ and $\lambda(e)$ should be dependent transitions and therefore $pre(e) \preceq e$ in the Tr. By reasoning similar to the one given in (a) part, we can deduce that there is no event e' that can change the fact d and $pre(e) \preceq e' \preceq e$, and $pre(e) = max^d_{\preceq}(\downarrow e)$. Therefore by Definition 4, $d \in in^{Tr}(e)$, since $d \in D^*_{\lambda(e)}$.
   (ii) $pre(e)$ is just passing d along without changing it, and therefore $d \in in^{\sigma}(pre(e))$. Then we just keep going back in σ until we reach an event e' such that e' generates d for the first time. Since e' generates d, we have $d \in D_{\lambda(e')}$ and by reasoning similar to the (i) case, we have $e' \preceq e$ in Tr and there is no e'' such that e'' changes d and $e' \preceq e'' \preceq e$, and therefore $e' = max^d_{\preceq}(\downarrow e)$, and $d \in in^{Tr}(e)$.

The above lemma leads to the proof of Theorem 1:

Proof. (sketch)

(a) $d \in MOT(t) \Rightarrow d \in MOP(t)$:
$d \in MOT(t)$ implies that there is a trace Tr and an event $e \in Tr$, $\lambda(e) = t$ such that $d \in in^{Tr}(e)$. Since $d \in in^{Tr}(e)$, for all linearizations σ of Tr, we have $d \in in^{\sigma}(e)|_{D^*_{\lambda(e)}}$ by Lemma 1.

(b) $d \in MOP(t) \Rightarrow d \in MOT(t)$:
$d \in MOP(t)$ implies that there is a run σ and an event $e \in \sigma$ where $\lambda(e) = t$ such that $d \in in^{\sigma}(e)|_{D^*_{\lambda(e)}}$. Consider the corresponding trace of σ, $Tr(\sigma)$; clearly, $Tr(\sigma)$ is a valid trace of the program. By Lemma 1, we know that $d \in in^{Tr(\sigma)}(e)$, and therefore $d \in MOT(t)$.

4.3 Backward Flow Analysis

In the presence of concurrency, backward flow analysis cannot be handled as a trivial dual of the forward analysis, as it is handled in the sequential case. The main reason for this, is that in contrast to the sequential case, not every position in the control net is reachable; presence of synchronization mechanisms such as locks can make certain positions unreachable.

Consider a class of CCD problems in which the transformation functions in $\mathcal{F}$ are interpreted backward, meaning that based on the facts that hold after the execution of a transition t, they return the facts that hold before the execution of t. To distinguish backward case form the forward case, we call these problems bCCD (backward Causal Concurrent Dataflow) problems. Here, we discuss the general MOT solution for bCCD problems.

Definition 8. *bTr is a backward trace of a control net N if and only if there is a* finite *trace $Tr = (\mathcal{E}, \preceq, \lambda)$ of N and $bTr = (\mathcal{E}, \succeq, \lambda)$ where $e_1 \succeq e_2 \Leftrightarrow e_2 \preceq e_1$.*

It is easy to see that bTr is a trace if Tr is a trace. Now, using the backward trace bTr and the trace solution of Definition 6, we can define the set of dataflow facts that hold before/after each event e in a trace Tr.

Definition 9. *For a bCCD problem $(N, \mathcal{S}, \mathcal{F})$ and for a trace $Tr = (\mathcal{E}, \preceq, \lambda)$ of the control net and for each event $e \in \mathcal{E}$, we define the following set of dataflow facts:*

$$\begin{cases} out^{Tr}(e) = in^{bTr}(e) \\ in^{Tr}(e) = \ f_{\lambda(e)}(out^{Tr}(e)) \end{cases}$$

where $in^{bTr}(e)$ is defined according to Definition 4, assuming all the maximal events of Tr (minimal events of bTr) have an initial value of $\perp$.

4.4 Formulation of Specific Problems in the CCD Framework

A wide variety of dataflow analysis problems can be formulated using the CCD framework, including reaching definitions, uninitialized variables, live variables, available expressions, copy constant propagation, very busy expressions, etc.

Some of these are *backward flow analysis* problems that can be formulated using an adaptation of CCD for backward flows. Due to lack of space, we detail only a couple of representative forward flow problems here; formulation of several others, including formulation of backward flows can be found in [23].

Reaching Definitions. The reaching definitions analysis determines: "*For each control point, which relevant assignments may have been made and not overwritten when program execution reaches that point along some path* ". The relevant assignments are the assignments to variables that are referred to in that control point. Given the control net $N = (P, T, F)$ for a program Pr, define $Defs = \{(v, t) \mid t \in T, v \in (\text{GVar} \cup \text{LVar}),$ and v is assigned in $t\}$. The property space is $(Defs, \subseteq, \cup, \emptyset)$, where presence of (v, t) in $D^{in}(t')$ means that the definition of v at t may reach t'.

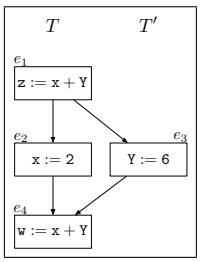
Let $D_t = \{(v, t') \mid v$ is assigned in $t\}$; $D_t^* = \{(v, t') \mid v$ is assigned or accessed by $t\}$.

For each transition t and each set $S \subseteq D_t$:

$$f_t(S)(= \begin{cases} S & \text{if } t \text{ is not an assignment} \\ S - \{(v, t') | t' \in T\} \cup \{(v, t)\} & \text{if } t \text{ is of the form v := e} \end{cases}$$

The construction of the control net ensures that two accesses of a variable v where one of them is a write, are dependent (neighborhoods intersect). This guarantees that the condition (*) of Definition 3 holds, i.e. our formulation of reaching-definitions ensures that information is inherited only from causal predecessors. Note that the above formulation is also distributive.

Available Expressions. The available expressions analysis determines: "*For a program point containing* $x := Exp(x_1, \ldots, x_k)$ *whether Exp has already been computed and not later modified on all paths to this program point*".



In the standard (sequential) formulation of available expressions analysis, dataflow facts are defined as pairs (t, Exp), where Exp is computed at t. This formulation does not work for the concurrent setting. To see why consider the trace on the right where `x` is a local variable in T and `Y` is a global variable. Events e_2 and e_3 are independent (concurrent), but they both can change (kill) the dataflow fact associated with `x + Y`, which is not in accordance with the condition (*) of Definition 3. The natural remedy is to divide this fact into two facts, one for `x` and another for `Y`. Let us call these two facts `x + Y : x` and `x + Y : Y`. The fact `x + Y : x` (respectively `x + Y : Y`) starts to hold when the expression `x + Y` is computed, and stops to hold when a definition to `x` (respectively `Y`) is seen. The problem is that `x + Y` holds when `x + Y : x` holds **and** `x + Y : Y` holds, which makes the framework *non-distributive*. Although we can solve non-distributive problems in the CCD framework (see Appendix), distributive problems yield faster algorithms (see Section 5).

The analysis can however be formulated as a distributive CCD problem by looking at the dual problem; that is, for *unavailability* of expressions. The dataflow fact $\mathtt{x+Y}$ indicates the expression being unavailable, and accordingly the presence of $\mathtt{x+Y}$: $\mathtt{x}$ **or** $\mathtt{x+Y}$: $\mathtt{Y}$ can make it hold. We are now in a distributive framework. Assume EXP presents the set of all expressions appearing in the program code, and define $\mathbb{D} = \{exp : x_i \mid exp \in EXP \ \wedge \ x_i \text{ appears in } exp\}$. The property space is the subset lattice $(\mathbb{D}, \subseteq, \cup, \mathbb{D})$, where presence of exp in $D^{in}(t')$ means that exp is unavailable at t. We have $D_t = D_t^* = \{exp : x \mid x \text{ is assigned in } t \text{ or } exp \text{ appears in } t\}$. For each transition t and each set $S \subseteq \mathbb{D}$:

$$f_t(S) = \begin{cases} S & t \text{ is not an assignment} \\ S \cup \ \{exp' : x \mid \forall exp' \in EXP, x \in V(exp')\} & \\ \quad - \{exp : y \mid y \in V(exp)\} & t \text{ is } x := exp \end{cases}$$

where $V(exp)$ denotes the set of variables that appear in exp.

Uninitialized Variables. The uninitialized variables analysis determines:

> For each relevant program point, which variables may be read before having previously been initialized when the program execution reaches that point.

Given the control net $N = (P, T, F)$ for a program Pr, define $Vars = GVar \cup LVar$. Formulation of uninitialized variables analysis in the CCD framework is as follows:

- The property space is $(Vars, \supseteq, \cap, Vars)$, where presence of v in $D^{in}(t')$ means that v may be uninitialized at t'. Note that at the beginning, everything is uninitialized.
- For each transition t and each set $S \subseteq Vars$:

$$f_t(S)(= \begin{cases} skip & t \text{ is not an assignment} \\ S - \{v\} & t \text{ is } v := e(v_1, \ldots v_n) \wedge \ \nexists i : v_i \in S \\ S \cup \{v\} & t \text{ is } v := e(v_1, \ldots v_n) \wedge \exists i : v_i \in S \end{cases}$$

  which makes $D_t = \{v \mid v \text{ is written in } t\}$.

In this problem, one starts with all variables being uninitialized. If t is an assignment $v := e(v_1, \ldots v_n)$ with some uninitialized argument v_i then v is added to the set of uninitialized variables, otherwise, v is removed from that set. Any transition t' that can affect the status of v_i must contain an assignment to it. This creates a similar setting to that of the reaching definitions problem since a write to v_i and a read of v_i (in t) are always ordered in any trace. Therefore, again considering the flow of facts along the causal edges suffices. Note that

the transformation functions are not monotonic in this case, and this is not a bit-vector problem, but since the information flows through the causal edges, it belongs to the CCD framework. Moreover, the transformation functions are distributive.

Live Variables A variable v is live at the exit point a definition to v if there is path from that point to a use of v which does not contain any definitions to v. The live variables analysis determines:

> For each definition of a variable v, wether v is live at the exit point of this definition.

Note that this is a variant of the standard textbook definition of live variables analysis, since the standard definition evaluates the liveness of a variable at any program point, not just the relevant ones (the definitions points). We argue that the above version is more sensible for concurrent programs since, if a thread T does not access global variable v at all, then information on liveness of v should not be relevant to an instruction in thread T.

Given the control net $N = (P, T, F)$ for a program Pr, define $\mathit{Vars} = \mathit{GVar} \cup \mathit{LVar}$. The live variables problem can then be summarized as:

- The property space is $(\mathit{Vars}, \subseteq, \cup, \emptyset)$, where presence of v in $D^{out}(t')$ means that v is live at the exit from t'.
- For each transition t and each set $S \subseteq \mathit{Vars}$:

$$f_t(S)(= \begin{cases} \mathit{skip} & t \text{ is not an assignment} \\ S - \{v\} & t \text{ is an assignment to } v \\ S \cup \{v\} & v \text{ is a used but not defined in } t \end{cases}$$

  which makes $D_t = \{v \mid v \text{ is accessed in } t\}$.

As argued in the previous cases, since variables liveness is changed by definitions and uses of the variable and the control net imposes all the definitions and uses to be causally related, our notion of information flowing through causal edges holds for this problem.

Copy Constant Propagation Copy constant propagation analysis determines:

> For a program point with a use of a variable v, whether or not v has a constant value whenever execution reaches that point.

Copy constant propagation analysis is a subproblem of the above definition in which only assignments of the form $\texttt{v} := \texttt{C}$ for C being an integer constant are taken into account for the computation of the constants. Any other form of assignment to a variable v is assumed to generate a non-constant ($\top$) value.

Given the control net $N = (P, T, F)$ for a program Pr, define $\mathit{Vars} = \mathit{GVar} \cup \mathit{LVar}$. Also, assume Z is set of all the constants appearing in the program code (which is finite assuming the code is finite) plus $\top$ which intuitively represents non-constant. Consider the partial order $\sqsubseteq$ on Z defined as follows:

$$\begin{aligned} \forall z \in Z :\ & z \sqsubseteq \top \\ \forall z, z' \in Z :\ & z \sqsubseteq z' \Leftrightarrow z = z' \end{aligned}$$

which naturally requires $z \sqcup z' = \top \Leftrightarrow z \neq z'$. A function $\sigma : \mathit{Vars} \to Z$ is called a substitution. Let Σ be set of all possible substitutions for Vars over Z.

The live variables problem can then be summarized as:

- The property space is $(\Sigma, \sqsubseteq, \sqcup, \bot)$, where presence of (v, c) in $D^{in}(t')$ means that v has a constant value of c at t'. Operations $\sqsubseteq, \sqcup$ are defined as follows:

$$\begin{aligned} \forall \sigma \in \Sigma :\ & \bot \sqsubseteq \sigma \\ \forall \sigma, \sigma' \in \Sigma :\ & \sigma \sqsubseteq \sigma' \ \Leftrightarrow\ \forall v \in \mathit{Vars} : \sigma(v) \sqsubseteq \sigma'(v) \\ \forall \sigma \in \Sigma :\ & \bot \sqcup \sigma = \sigma \\ \forall \sigma, \sigma' \in \Sigma :\ & \forall v \in \mathit{Vars} : (\sigma \sqcup \sigma')(v) = \sigma(v) \sqcup \sigma'(v) \end{aligned}$$

- For each transition t and each set $S \subseteq \mathbb{D}$:

$$f_t(\sigma)(= \begin{cases} \mathit{skip} & t \text{ is not an assignment} \\ \sigma[v \leftarrow c] & t \text{ is } v := c \\ \sigma[v \leftarrow \top] & t \text{ is any other assignment} \end{cases}$$

As argued in the previous, since variables use points are causally related to the assignment statements that indicate the value of constants, our notion of information flowing through causal edges holds for this problem.

Very Busy Expressions An expression is very busy at the exit from a transition computing it if, on all paths starting from this transition the expression in used before any of the variables in it is redefined. The very busy expression analysis determines:

For a assignment point $\texttt{x} := \texttt{exp}$ wether $\texttt{exp}$ is a very busy expression.

Similar to the available expressions framework, we look at the dual problem, and the question whether `exp` is not very busy. The dataflow fact $(t, \texttt{exp}) : \texttt{x}_\texttt{i}$ means that An expression $mathttexp(x_1, \ldots, x_n)$ is not very busy at the exit point from an assignment if, there is x_i that is redefined before it is used (the use refers to the later appearance of same expression `exp`).

- The property space is the subset lattice $(\mathbb{D}, \subseteq, \cup,)$, where presence of $exp : \texttt{x}$ (for all $\texttt{x}$) in $D^{in}(t')$ means that exp is unavailable at t.
- For each transition t and each set $S \subseteq EXP$:

$$f_t(S)(= \begin{cases} skip & t \text{ is not an assignment} \\ S \quad -\{exp : x\} & t \text{ is } x := exp' \\ S \quad \cup\{exp : x \mid x \in exp\} & t \text{ is } v := exp \end{cases}$$

5 Solving the Distributive CCD Problem

In this section, we show how to solve a dataflow problem in the CCD framework. The algorithm we present is based on augmenting a control net to a larger net based on the dataflow analysis problem, and reducing the problem of checking whether a dataflow fact holds at a control point to a reachability problem on the augmented net. The augmented net is carefully constructed so as to not destroy the concurrency present in the system (crucially exploiting the condition (*) in Definition 3). Reachability on the augmented net is performed using net unfoldings, which is a partial-order based approach that checks traces generated by the net as opposed to checking linear runs.

Due to space restrictions, we present only the solution for the distributive CCD problems where the meet operator is union, and we prove upper bounds that compare the unfolding of the augmented net with respect to the size of the unfolding of the original control net.

In order to track the dataflow facts, we enrich the control net so that each transition performs the transformation of facts as well. We introduce new places which represent the dataflow facts. The key is then to model the transformation functions, for which we use *representation relations* from [6].

Definition 10. *The representation relation of a distributive function $f : 2^D \rightarrow 2^D$ ($D \subseteq \mathbb{D}$) is $R_f \subseteq (D \cup \{\bot\}) \times (D \cup \{\bot\})$, a binary relation, defined as follows:*

$$R_f = \{(\bot, \bot)\} \ \cup \ \{(\bot, d) \mid d \in f(\emptyset)\} \ \cup \ \{(d, d') \mid d' \in f(\{d\}) \wedge \ d' \notin f(\emptyset)\}$$

The relation R_f captures f faithfully in that we can show that $f(X) = \{d' \in D \mid (d, d') \in R_f, \ \textit{where } d = \bot \textit{ or } d \in X\}$, for any $X \subseteq D$.

Given a CCD framework $(N, \mathcal{S}, \mathcal{F}, \mathcal{D}, \mathcal{D}^*)$ with control net $N = (P, T, F)$, we define the net representation for a function f_t as below:

Definition 11. *The net representation of f_t is a Petri net $N_{f_t} = (P_{f_t}, T_{f_t}, F_{f_t})$ defined as follows:*

- *The set of places is $P_{f_t} = {}^\bullet t \cup t^\bullet \cup \{\bot_m \mid m \in [1,n]\} \cup \bigcup_{d_i \in D_t} \{p_i, \overline{p}_i\}$ where a token in p_i means the dataflow fact d_i holds, while a token in $\overline{p}_i$ means that d_i does not hold, and n is the number of dataflow facts.*
- *The set of transitions T_f contains exactly one transition per pair $(d_i, d_j) \in R_{f_t}$, and is defined as:*

$$T_{f_t} = \left\{ s^t_{(\bot,\bot)} \right\} \cup \left\{ s^t_{(\bot,j)} |\ (\bot, d_j) \in R_{f_t} \right\} \cup \left\{ s^t_{(i,j)} |\ (d_i, d_j) \in R_{f_t} \right\}$$

  Note that if $D_t = \emptyset$ then $T_{f_t} = \left\{ s^t_{(\bot,\bot)} \right\}$.
- *The flow relation is defined as follows:*

$$\begin{aligned} F_{f_t} = & \bigcup_{s \in T_{f_t}} \left(\bigcup_{p \in {}^\bullet t} \{(p,s)\} \ \cup \bigcup_{p \in t^\bullet} \{(s,p)\} \right) \cup \bigcup_{d_k \in D_t} \left\{ (\overline{p}_k, s^t_{(\bot,\bot)}), (s^t_{(\bot,\bot)}, \overline{p}_k) \right\} \\ & \cup \bigcup_{(\bot, d_j) \in R_{f_t}} \left(\left\{ (\bot_m, s^t_{(\bot,j)}) \mid t \in T_m \right\} \cup \left\{ (s^t_{(\bot,j)}, p_j) \right\} \right. \\ & \qquad \left. \cup \bigcup_{d_k \in D_t} \left\{ (\overline{p}_k, s^t_{(\bot,j)}) \right\} \cup \bigcup_{k \neq j} \left\{ (s^t_{(i,j)}, \overline{p}_k) \right\} \right) \\ & \cup \bigcup_{\substack{(d_i, d_j) \in R_{f_t} \\ i \neq j}} \left(\left\{ (p_i, s^t_{(i,j)}), (s^t_{(i,j)}, p_j), (\overline{p}_j, s^t_{(i,j)}), (s^t_{(i,j)}, \overline{p}_i) \right\} \right) \\ & \cup \bigcup_{(d_i, d_i) \in R_{f_t}} \left(\left\{ (p_i, s^t_{(i,i)}), (s^t_{(i,i)}, p_i) \right\} \right) \end{aligned}$$

The idea is that each transition $s^t_{(i,j)}$ is a copy of transition t that, besides simulating t, models one pair (d_i, d_j) of the relation R_{f_t}, by taking a token out of place p_i (meanwhile, also checking that nothing else holds by taking tokens out of each $\overline{p}_k$, $k \neq i$) and putting it in p_j (also returning all tokens $\overline{p}_k$, $k \neq j$). Thus if d_i holds (solely) before execution of t, d_j will hold afterwards. The transitions $s^t_{\bot,j}$ generate new dataflow facts, but consume the token $\bot_m$ associated with the thread. We will engineer the net to initially contain only one $\bot_m$ marking (for some thread m), and hence make sure that only one fact is generated from $\bot$.

For every t, transitions $s^t_{(i,j)}$ are in *conflict* since they have ${}^\bullet t$ as common predecessors. This means that only one of them can execute at a time, generating a single fact. If we assume that initially nothing holds (i.e., initial tokens are in every $\overline{p}_i$'s and no initial tokens in any of the p_i's), then since each transition consumes one token and generates a new token, the following invariant always holds for the system: "At any reachable marking of the augmented net, exactly one position p_i corresponding to some dataflow fact d_i holds". We use this observation later to argue the complexity of our analysis.

Definition 12. *The augmented marked net $N^{\mathcal{S},\mathcal{F}}$ of a CCD problem $(N, \mathcal{S}, \mathcal{F})$ is defined as $\bigcup_{f\in\mathcal{F}} N_f$ where the union of two nets $N_1 = (P_1, T_1, F_1)$ and $N_2 = (P_2, T_2, F_2)$ is defined as $N_1 \cup N_2 = (P_1 \cup P_2, T_1 \cup T_2, F_1 \cup F_2)$. It is assumed that N_f's have disjoint set of transitions, and only the common places are identified in the union. Furthermore we add a new position p^*, make each $\bar{p}_i$ initial, and also introduce n initial transitions t^*_m, one for each thread, that removes p^* and puts a token in $\bot_m$ and a token in the initial positions of each thread.*

The above construction only works when $\bot = \emptyset$. When $\bot = D_0$, for some $D_0 \subseteq \mathbb{D}$, we will introduce a new initial set of events (all in conflict) that introduce nondeterministically a token in some $p_i \in D_0$ and remove $\bar{p}_i$.

Example: Consider the reaching definitions problem (which is distributive) for the program on the right and assume we are interested in the global variable `Y`. Figure 6(a) illustrates the control net of the program. There are two definitions of Y at t_1 and t_2, therefore $\mathbb{D} = \{(\mathtt{Y}, t_1), (\mathtt{Y}, t_2)\}$. $f_{t'_1}$ is the skip function. The representation relation for functions f_{t_1} and f_{t_2} are as in Figure 5.

```
   T                        T'

Y := 5;                      ||
Y := 3;            x := Y - 2;
```

Figures 6(b,c) present the net representations of the above relations, considering that initially $\bot$ holds. Figure 6(d) shows the union of the three nets in Figures 6(a,b,c) which is the augmented net of program for the reaching definitions problem.

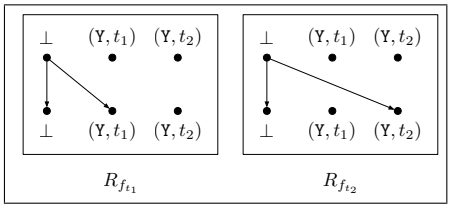


Fig. 5. Representation relation.

The problem of computing the *MOT* solution can be reduced to a *coverability* problem on the augmented net. To be more precise, fact d_i *may* hold before the execution of transition t of the control net if and only if $\{p_i, p_t\}$ is coverable from the initial marking of the control net where p_t is the local control place associated to transition t in its corresponding thread.

Example: Consider the example from Figure 6. In the augmented net $\{p_\bot\} \cup {}^\bullet t_1 = \{p_\bot, p_1, p_{\mathtt{Y}_1}, p_{\mathtt{Y}_2}\}$ is coverable. which means $\bot$ holds before the execution of t_1. Similarly, $\{p_{(\mathtt{Y},t_1)}\} \cup {}^\bullet t_2 = \{p_{(\mathtt{Y},t_1)}, p_2, p_{\mathtt{Y}_1}, p_{\mathtt{Y}_2}\}$ which means $(\mathtt{Y}, t_1)$ holds before the execution of t_2. It is also easy to see that $\{p_{(\mathtt{Y},t_1)}\} \cup {}^\bullet t_1 = \{p_{(\mathtt{Y},t_1)}, p_1, p_{\mathtt{Y}_1}, p_{\mathtt{Y}_2}\}$ is not coverable (to have a token in $p_{(\mathtt{Y},t_1)}$, a token from p_1 must be consumed) and therefore $(\mathtt{Y}, t_1)$ does not hold before execution of t_1.

Theorem 2. *A dataflow fact d_i holds before the execution of a transition t in the control net N of a program if and only if $d_i \in D^*_t$ and the marking $\{p_i, p_t\}$ is coverable from the initial marking in the augmented net $N^{\mathcal{S},\mathcal{F}}$ constructed according to Definition 12.*

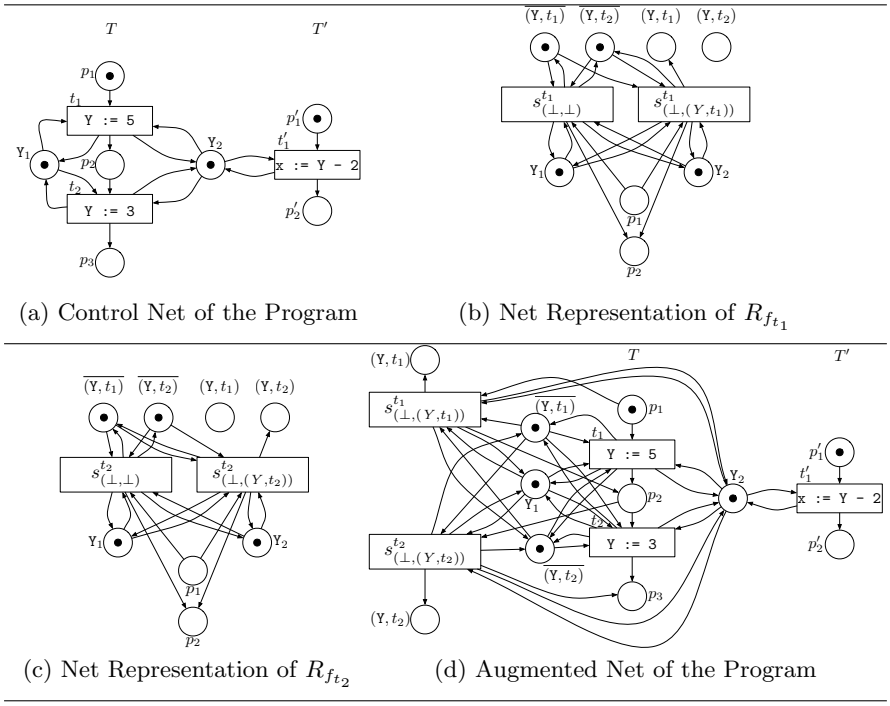


Fig. 6. Augmented Control Net.

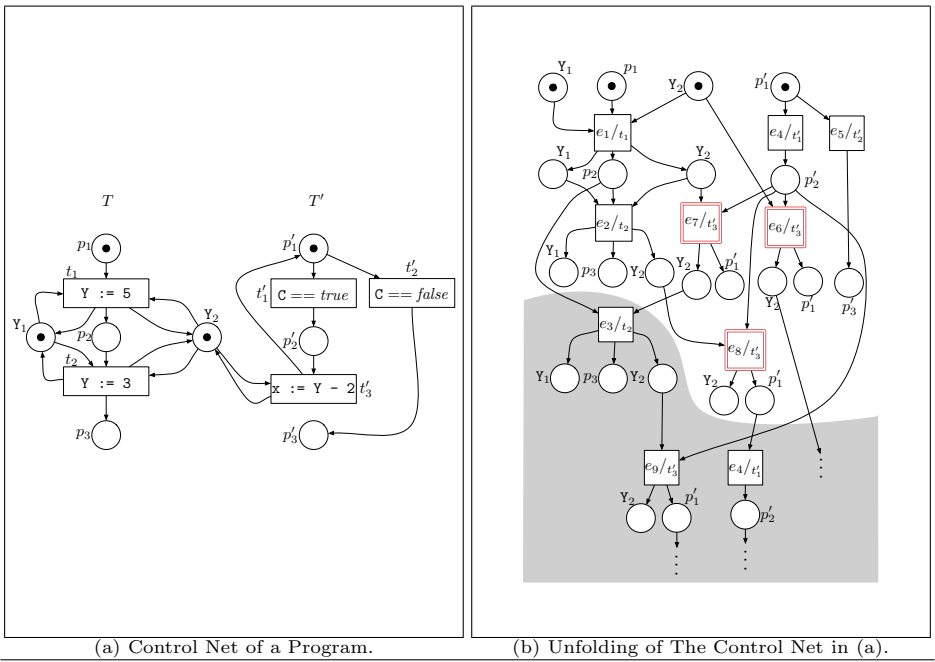
Checking coverability: While there are many tools that can check reachability/coverability properties of Petri nets, tools that use *unfolding* techniques [7, 8] of nets are particularly effective, as they explore the state space using partially ordered unfoldings and give automatic reduction in state-space (akin to partial-order reduction for model checking of concurrent systems). We assume the reader is familiar with net unfoldings and refer to [8] for details.

Complexity of distributive CCD: Algorithms for Petri nets which use finite unfoldings essentially produces a finite unfolding of the net, from which coverability of one position can be checked in linear time. For every transition $t' \in T_{f_t}$ and every fact $d_i \in D_t^*$, we can create a new transition whose preconditions are those of t' plus p_i, and outputs a token in a new position (t, d_i). By Theorem 2, coverability of this single position is equivalent to fact d_i holding at t. Furthermore, we can argue that the unfolding of this net introduces at most $n|\mathbb{D}|$ new events compared to the unfolding of the augmented net.

Let us now analyze the size of the unfolding of the augmented net in terms of the size of the unfolding of the original control net; let us assume the latter has n events. We can show that (a) every marking reachable by a local configuration of the control net has a corresponding event in its finite unfolding that realizes this marking, and (b) that for every marking reached by a local configuration of the control net, there are at most $|\mathbb{D}|$ corresponding local configurations in the augmented net (at most one for each dataflow fact), and this covers all local configurations of the augmented net. Since the number of events in the unfolding is bounded by the number of markings reachable by local configurations, it follows that the size of the unfolding of the augmented net is at most $|\mathbb{D}|$ times that of the control net. This argues the efficacy of our approach in preserving the concurrency inherent in the control net and in exploiting distributivity to its fullest extent.

5.1 Net Unfoldings

Consider the control net in Figure 7(a). Places and transitions have unique labels, and the initial marking is $\{p_1, p_1', Y_1, Y_2\}$. To build the unfolding of this net (Figure 7(b)), we start by the set of places in the initial marking. At each step, a copy of transition t, for which the set of places in ${}^\bullet t$ are available, is added to the unfolding, and the set set of places in $t^\bullet$ are as a result added. We call these copies *events*. These events are labeled by their corresponding transitions. Note for each place p in $t^\bullet$, a new place is added to the unfolding even if p already existed. For example, transition t_1 can be added after the initial marking, and the places p_2, Y_1, Y_2 are added after that, although Y_1 and Y_2 already existed as part of the initial marking. The only constraint for adding a transition t is that the set of places in ${}^\bullet t$ should all be concurrent; that is not causally related and not in conflict. Also, the same set of places cannot be used more than once



(a) Control Net of a Program. (b) Unfolding of The Control Net in (a).

Fig. 7. Control Net Construction

to generate the same transition. This way, the unfolding for the control net in Figure 7(a) is infinite, part of which is illustrated in Figure 7(b).

An interesting property of the net unfoldings is that, it contains (exactly) the set of all reachable markings of the original net [7, 8]. Therefore, all the queries of coverability/reachability can be answered on the unfolding instead of the net itself.

Another interesting property of net unfoldings is that if the original Petri net is *1-safe*[6], then there exists a *finite prefix* of the unfolding of the net which contains all the reachable markings of the original net [7, 8]. It is also shown in [8] that this finite prefix has size $O(n)$ where n is the number of reachable markings of the original net. In Figure 7(b), the unshaded area contains the finite prefix of the unfolding of the net in Figure 7(a). The double-lined squares indicate the so-called *cut-off* events which mark the boundaries of the finite prefix.

Theorem 3. *Let* $(N, \mathcal{S}, \mathcal{F})$ *be a distributive CCD problem, with* $\mathcal{S} = (\mathcal{P}(\mathbb{D}), \subseteq, \cup, \bot)$. *Let* n *be the size of the unfolding of* N. *Then the size of the unfolding*

[6] A Petri net is 1-safe if no place can contain more than one token in it. Control nets are by definition 1-safe.

of the augmented net $N^{\mathcal{S},\mathcal{F}}$ (and even the complexity of checking whether a fact holds at a control point) is at most $O(n|\mathbb{D}|)$.

5.2 Augmented Net for Non-distributive Problems

In the case of non-distributive frameworks, the singletons are not sufficient for modeling the transformation functions. Therefore, the transformation functions have to be defined for all elements of 2^{D_t} for each function f_t.

Definition 13. *Representation relation of a non-distributive $f : 2^D \to 2^D$ ($D \subseteq \mathbb{D}$), $R_f \subseteq 2^D \times 2^D$ is a binary relation defined as follows:*

$$R_f^{nd} = \{(S, S') \mid S, S' \subseteq D \ \wedge \ f(S) = S'\} \cup \{(\emptyset, \emptyset)\}$$

Given an CCD problem with the property space $(\mathbb{D}, \sqsubset, \sqcup, \perp)$ where $\mathbb{D} = \{d_1, \ldots, d_m\}$ and with control net $N = (P, T, F)$, we define the net representation for the set of transformation functions $\{f_t\}_{t \in T}$:

Definition 14. *Net representation of f_t is a Petri net $N_{f_t} = (P_{f_t}, T_{f_t}, F_{f_t})$ defined as follows:*

- *The set of places is defined as $P_{f_t} = \bigcup_{d_i \in D_t} \{p_i, \overline{p}_i\}$ where a token in place p_i means the dataflow fact d_i holds, while a token in $\overline{p}_i$ means that d_i does not hold.*
- *The set of transitions T_f, which contains exactly one transition per pair $(d, d') \in R_{f_t}$, is defined as:*

$$T_{f_t} = \left\{ s^t_{(\emptyset,\emptyset)} \right\} \cup \left\{ s^t_{(S,S')} |\ (S, S) \in R^{nd}_{f_t} \right\}$$

- *The flow relation is defined as follows:*

$$\begin{aligned} F_{f_t} = & \bigcup_k \left\{ (\overline{p}_k, s^t_{(\emptyset,\emptyset)}), (s^t_{(\emptyset,\emptyset)}, \overline{p}_k) \right\} \cup \bigcup_{s \in T_{f_t}} \left(\bigcup_{p \in {}^\bullet t} \{(p, s)\} \ \cup \ \bigcup_{p \in t^\bullet} \{(s, p)\} \right) \\ & \cup \bigcup_{(S,S') \in R^{nd}_{f_t}} \left(\bigcup_{d_i \in S} \left\{ (p_i, s^t_{(S,S')}) \right\} \cup \bigcup_{d_j \in D_t - S} \left\{ (\overline{p}_j, s^t_{(S,S')}) \right\} \ \cup \right. \\ & \left. \bigcup_{d_i \in S'} \left\{ (s^t_{(S,S')}, p_i) \right\} \cup \bigcup_{d_j \in D_t - S'} \left\{ (s^t_{(S,S')}, \overline{p}_j) \right\} \right) \end{aligned}$$

Definition of the augmented net remains the same as Definition 12 by replacing $\perp$ with $\emptyset$. Similar to the distributive case, a fact d_i holds at the entry point of a transition t if and only if ${}^\bullet t \cup \{p_i\}$ is coverable from the initial marking, and Theorem 2 states the correctness of the approach. Also, similar reasoning as Theorem 3 can show that the time/space complexity bound on the size of the unfolding in this case is $2^{|\mathbb{D}|} \times |U_N|$ where U_N is the unfolding of the program control net.

5.3 Augmented Net for Backward Flow Problems

Here, we present the construction of augmented net for the backward flow only for the distributive framework. In the backward flow problems, the transformation functions are interpreted in the reverse direction, in the sense that $D_{in}(e) = f_{\lambda(e)}(D_{out}(e))$ as opposed to $D_{out}(e) = f_{\lambda(e)}(D_{in}(e))$. We work with the inverse of these functions (that are not necessarily functions) for the construction of the augmented net. It is easy to see that for each f_t, $R_{f_t}^{-1}$ (for R_{f_t} defined as in Definition 10) models the relation f_t^{-1}.

The backward analysis is more tricky than the forward case. Assume we want to check whether $d \in D_{out}(t)$. By Definition 9, d holds at this point because there is a trace Tr (let us call this the witness trace for d) with events e, e' such that $e \preceq e'$ and e' generates d and no event e'' ($e \preceq e'' \preceq e'$) changes d.

The first step is to make sure that t is reachable; if t is not reachable, there is no interest in dataflow facts that may reach it[7]. After checking reachability of t, the next step is to indicate that we intend to check wether d holds at the exit from t; since different facts may have different witness traces.

Definition 15. *A (t, d_i)-monitoring net $N_{(t,d_i)}$ for a transition t of a program control net N and a dataflow fact $d_i \in D_t = \{d_1, \ldots, d_m\}$, is a small Petri net with one transition t_{d_i}, and one place p_i^* such that ${}^\bullet t_{d_i} = {}^\bullet t \cup \{\overline{p}_1, \ldots, \overline{p}_m\}$ and $t_{d_i}^\bullet = t^\bullet \cup \{p_i^*, p_i\} \cup \bigcup_{k \neq i} \{d_k\}$.*

The purpose of a (t, d_i)-monitor is to indicate (by putting a token in place p_i^* which is initially empty) that d_i is the fact to be checked to be in $D_{out}(t)$. t_{d_i} is a special copy of t that is enabled when t is enabled and no dataflow fact holds. It then assumes d_i holds by putting a token in p_i.

The main difference between the backward case and the forward case (introduced earlier) is that, in the backward analysis, the augmented net is specialized to check for a specific fact at a specific point.

[7] Note that in the concurrent setting, in contrast to the sequential case, some transitions may not be reachable.

Definition 16. *The augmented net of a backward CCD problem $(N, \mathcal{S}, \mathcal{F})$ to check for a data flow fact d to hold at the exit from transition t, is defined as $N \cup \bigcup_{f \in \mathcal{F}} N_{f^{-1}} \cup N_{(t,d)}$ where the union of two nets $N_1 = (P_1, T_1, F_1)$ and $N_2 = (P_2, T_2, F_2)$ is defined as $N_1 \cup N_2 = (P_1 \cup P_2, T_1 \cup T_2, F_1 \cup F_2)$ with the exception of identifying each transition t of N with transitions $s^t_{(\bot,\bot)}$ of N_is. It is assumed that N_fs have disjoint set of transitions, and only the common places are identified in the union.*

Note that although the above definition of the augmented net for the backward case is specific to a fact d and a transition t, one can always generalize it by adding (by union) all the (t, d_i)-monitors such that $d_i \in D_t$.

Theorem 4. *A dataflow fact d_i holds at the exit from a transition t in the control net N of a program if and only if $d_i \in D^*_t$ and the marking $\{p^*_i\} \cup \{\overline{p}_1, \ldots, \overline{p}_m\}$ (for $D_t = \{d_1, \ldots, d_m\}$) is coverable from the initial marking in the augmented net constructed according to Definition 16.*

Proof. We skip this proof since it is tedious and the ideas behind it are more or less the same as those offered in the proof of Theorem 2.

6 Experiments

We have applied the techniques from Section 5 to perform several dataflow analyses for concurrent programs. Unfortunately, there is no standard benchmark for concurrent dataflow programs. We have however experimented our algorithms with sample programs for the primary dataflow analysis problems, and studied performance when the number of threads is increased.

The motive of the experiments is to exhibit in practice the advantages of concurrent dataflow that exploit the causal framework set forth in this paper. While the practical efficacy of our approach on large programs is still not validated, we believe that setting up a general framework with well-defined problems permitting reasonable algorithms is a first step towards full-scale flow analysis. Algorithms that work on large code may have to implement approximations and heuristics, and we believe that the our framework will serve as a standard for correctness.

In many of our examples, there is an exponential increase in the set of reachable states as one increases the number of threads, but the partial order methods inherent to these techniques substantially alleviate the problem. We use the PEP tool [9] to check the coverability property on the augmented net to answer the relevant coverability queries.

For each example, we have included the sizes of the unfolding for the program's control net and of the augmented net (see Table 1). The *construction* time refers

to the time to build the unfolding, and the *checking* time refers to the time for a single fact checking. Note the huge differences between the two times in some cases, and also note that the unfolding is only built once and is then used to answer several coverability queries. All experiments were performed on a Linux machine with a 1.7GHz processor and 1GB of memory. The numbers are all in seconds (with a precision of 0.01 seconds).

Example	$\|\mathbb{D}\|$	#Threads	Unfolding Control Net	Unfolding Augmented Net	Time Checking (sec)	Time Construction(sec)
UV(10)	11	11	906	4090	< 0.01	<0.01
UV(20)	21	21	3311	16950	< 0.01	0.70
UV(60)	61	61	40859	156390	0.01	60.11
RD(3)	4	6	410	1904	< 0.01	0.03
RD(4)	5	8	1545	9289	0.01	1.5
RD(5)	6	10	5596	41186	0.01	133.16
RDL(3)	6	4	334	1228	< 0.01	0.01
RDL(4)	8	5	839	3791	< 0.01	29
RDL(5)	10	6	2024	10834	< 0.01	5.35
RDL(6)	12	7	4745	29333	0.01	121.00
AE(50)	2	50	250	650	< 0.01	< 0.01
AE(150)	2	150	750	1950	< 0.01	0.34
AE(350)	2	350	1750	4550	< 0.01	4.10

Table 1. Programs and Performances

Uninitialized Variables. This set of examples contains a collection of n threads with n global variables $\mathtt{X}^0, \ldots, \mathtt{X}^n$. One uninitialized variable $\mathtt{X}^0$ in one thread can consequently make all $\mathtt{X}^i$s uninitialized. Concurrency results in many possible interleavings in this example, a few of which can make a certain variable $\mathtt{X}^j$ uninitialized.

Reaching Definitions. This example set demonstrates how our method can successfully handle synchronization mechanisms. There are two types of threads: (1) those which perform two consequent writes to a global variable `Y`, and (2) those which perform a read of `Y`. There are two variations of this example: (1) where none of the accesses is protected by a lock, which we call RD, and (2) where the read, and the two writes combined are protected by the same lock, which we call RDL (the code on the right). The main difference between the two versions is that $\mathtt{Y} := 1$ will reach the read in the lock-free version, but cannot reach it in the presence of the locks. In a setting with one copy of T' and n copies of T, there are $2n$ definitions where only n of them can reach the line $\mathtt{x} := \mathtt{Y} + 1$ of T'.

```
    T                T'

acquire(l);     acquire(l)
  Y := 1;         x := Y + 1;
  Y := 2;       release(l)
release(l);
```

Available Expressions. The example set AE shows how the unfolding method can fully benefit from concurrency. The threads here do not have any dependencies. Each thread defines the same expression $\mathtt{X} + \mathtt{Y}$ twice, and therefore, the

expression is always available for the second instruction of each thread. Table 1 shows that in the case of zero dependencies, the size of the unfolding grows linearly with the number of threads (understandably so since new threads do not introduce new dataflow facts).

7 Conclusions

The main contribution of this paper lies in the definition of a framework that captures dataflow analysis problems for concurrent program using partial orders that preserves the concurrency in the system. The preserved concurrency has been exploited in the partial-order based analysis, but could instead have been exploited in other ways, for example using partial-order reduction strategies as those used in SPIN.

As for future directions, the first would be to study *local* or *compositional* methods to solve the CCD problems and deploy them on large real world programs. This would have to handle (approximately) complex data such as pointers and objects. Our algorithms do not work for programs with *recursion*, and it is well known that dataflow analysis for concurrent programs with recursion quickly leads to undecidability. Structural restrictions like nested locking (see [24]) would be worth studying to obtain decidable fragments. Studying a framework based on computing *minimal fixpoints* for concurrent programs would be also interesting. Extending our approach to decide flow problems with *infinite domains of finite height* is challenging as well (they can be handled in the sequential setting [13]).

References

1. Hecht, M.: Flow Analysis of Computer Programs. Elsevier Science Inc. (1977)
2. Stoltz, E., Wolfe, M.: Sparse data-flow analysis for dag parallel programs (1994)
3. Lee, J., Midkiff, S.P., Padua, D.A.: Concurrent static single assignment form and constant propagation for explicitly parallel programs. In: Languages and Compilers for Parallel Computing. (1997) 114–130
4. Lee, J., Padua, D.A., Midkiff, S.P.: Basic compiler algorithms for parallel programs. In: Principles Practice of Parallel Programming. (1999) 1–12
5. Farzan, A., Madhusudan, P.: Causal atomicity. In: CAV. LNCS 4144 (2006) 315 – 328
6. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL. (1995) 49–61
7. McMillan, K.: A technique of state space search based on unfolding. Formal Methods in System Design **6**(1) (1995) 45–65
8. Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan's unfolding algorithm. Formal Methods in System Design **20** (2002) 285–310
9. Grahlmann, B.: The PEP tool. In: CAV. (1997) 440–443
10. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1986)
11. Nielson, F., Nielson, H.: Type and effect systems. In: Correct System Design. (1999) 114–136

12. Muchnick, S.S.: Advanced Compiler Design and Imlementation. Morgan Kaufmann (1997)
13. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. **58**(1-2) (2005) 206–263
14. Masticola, S.P., Ryder, B.G.: Non-concurrency analysis. In: PPOPP. (1993) 129–138
15. Naumovich, G., Avrunin, G.S.: A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In: SIGSOFT/FSE-6. (98) 24–34
16. Salcianu, A., Rinard, M.: Pointer and escape analysis for multithreaded programs. In: PPoPP, ACM Press (2001) 12–23
17. Knoop, J., Steffen, B., Vollmer, J.: Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. TOPLAS **18**(3) (1996) 268–299
18. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4) (1991) 451–490
19. Dwyer, M., Clarke, L., Cobleigh, J., Naumovich, G.: Flow analysis for verifying properties of concurrent software systems (2004)
20. Dwyer, M.B., Clarke, L.A.: A compact petri net representation and its implications for analysis. IEEE Trans. Softw. Eng. **22**(11) (1996) 794–811
21. Chamillard, A.T., Clarke, L.A.: Improving the accuracy of petri net-based analysis of concurrent programs. In: ISSTA, New York, NY, USA (1996) 24–38
22. Diekert, V., Rozenberg, G.: The Book of Traces. World Scientific Publishing Co. (1995)
23. Farzan, A., Madhusudan, P.: Causal dataflow analysis for concurrent programs. Technical Report UIUCDCS-R-2007-2806, CS Department, UIUC (2007)
24. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about thread communicating via locks. In: CAV. Volume LNCS 3576. (2005) 505–518