Inferring Synchronization under Limited Observability

Martin Vechev, Eran Yahav, and Greta Yorsh

IBM T.J. Watson Research Center

Abstract. This paper addresses the problem of automatically inferring synchronization for concurrent programs. Given a program and a specification, we infer synchronization that avoids all interleavings violating the specification, but permits as many valid interleavings as possible. We let the user specify an upper bound on the cost of synchronization, which may limit the *observability* — what observations on program state can be made by the synchronization code. We present an algorithm that infers, under certain conditions, the *maximally permissive* synchronization for a given cost. We implemented a prototype of our approach and applied it to infer synchronization in a number of small programs.

1 Introduction

Concurrency is hard. Concurrent execution of operations that share data requires synchronization to guarantee correctness. Typically, the programmer is required to reason about all the ways in which concurrent operations can interleave, and introduce synchronization code that avoids incorrect interleavings. Because of the excruciating difficulty in finding even a single choice of synchronization that makes the program correct and reasonably efficient [15], programmers often introduce synchronization in an ad-hoc manner, and rarely explore alternative choices. In particular, programmers often resort to coarse-grained synchronization because: (i) it simplifies reasoning about the program, and (ii) the overhead incurred by finer-grained synchronization is prohibitive.

Our goal is to assist the programmer in systematically exploring alternative choices of synchronization, based on the cost that she is willing to accept. Given a program P, and a specification S, we define the set $\mathit{VP}(P, S)$ of concurrent programs that satisfy S and can be obtained from P by adding synchronization. To understand the tradeoffs between different choices of synchronization code, we examine two dimensions along which programs in $\mathit{VP}(P, S)$ can be compared:

- **Permissiveness:** Given two programs $P_1, P_2 \in \mathit{VP}(P, S)$, we say that P_1 is *more permissive* than P_2 when the set of traces permitted by P_1 is a superset of the set of traces permitted by P_2.
- **Synchronization Cost:** Given two programs $P_1, P_2 \in \mathit{VP}(P, S)$, we say that P_1 has *lower cost* than P_2 when the running time of the synchronization code in P_1 is lower than that of P_2.

There is a connection between the cost of synchronization and its permissiveness. For the synchronization code to be more permissive, it needs to draw finer distinctions between interleavings, which typically requires atomically observing more of the program's state. Atomically observing more of the program's state means increasing the synchronization cost.

In general, the user would like to maximize permissiveness and minimize the cost. However, the synchronization solution that provides maximal permissiveness maybe too costly. There may be another (incomparable) solution, with less permissiveness and lower cost, which is acceptable. We let the user specify an upper bound on the cost, and infer a maximally permissive solution within the limits of this upper bound.

There are various synchronization mechanisms available to concurrent programmers today. In this paper we choose to focus on the classical conditional critical regions (CCRs), an elegant construct originally introduced by Hoare [7]. A CCR consists of a guard and a sequence of statements that are to be executed atomically if the guard evaluates to *true*. If the guard evaluates to *false*, the thread blocks until it is able to atomically re-evaluate the guard. Guards only observe program state, but cannot modify it. CCRs have been implemented as a synchronization construct in the language Edison [5], as a language extension of Java via software transactional memory [6], and recently in the high-performance parallel language X10 [14]. One of the advantages of CCRs over other lower-level operational primitives such as locks and condition variables is their concise and declarative nature.

A key challenge in using CCRs is finding the appropriate guard expressions. A programmer must address the following: (i) correctness — guards must eliminate invalid interleavings that violate S; (ii) permissiveness — guards should allow as many interleavings as possible: a thread executing a guard should not block unless its execution is doomed to violate S; (iii) cost — it is important to reduce the cost of evaluating the guard expression. Because the guard is evaluated atomically, this cost is typically dictated by the number of shared variables accessed in the guard. One way to reduce the cost is by restricting the code to observe only a subset of these variables. Balancing these trade-offs may require the programmer to simultaneously consider all guards in all of the CCRs in the program.

This work addresses the challenge of automatically inferring correct and maximally permissive guards, without exceeding the upper bound on the cost of the guards, specified by the user. This bound restricts the *language of guards* — the expressions that can be used as guards — to those that cost less than the specified bound.

Consider a concurrent program P, a specification S, and a language of guards *LG*. We denote by $\mathit{VP}(P, S, \mathit{LG})$ the set of programs that satisfy S and are obtained from P by adding guards from *LG*. It is possible that no program $P' \in \mathit{VP}(P, S, \mathit{LG})$ permits all valid interleavings of P. The reason is that the language *LG* may not be expressive enough to distinguish between a valid and an invalid interleaving and thus a valid program P' must avoid both. It is therefore natural to define the notion of a *maximally-permissive program* under a given language of guards: $P' \in \mathit{VP}(P, S, \mathit{LG})$ is maximally-permissive with respect to *LG* if there is no program in $\mathit{VP}(P, S, \mathit{LG})$ that permits more interleavings than P'. In other words, it is impossible to modify P' using expressions from *LG* to permit more interleavings without violating the specification S. Our goal in this paper can be stated as follows:

Given a concurrent program P, a specification S, and a language of guards LG, construct a program $P' \in \mathit{VP}(P, S, \mathit{LG})$, such that P' is maximally permissive with respect to LG, and has minimal synchronization cost.

The above problem statement is closely related to the ones addressed by program repair [9] and controller synthesis [12]. However, in contrast to these approaches, our work focuses on inferring synchronization code that observes the state without modifying it, and takes into account the cost of synchronization when attempting to find the maximally permissive solution.

1.1 Main Contributions

The contributions of this paper can be summarized as follows:

- We present a technique for automatically inferring correctly-synchronized concurrent programs. To explore alternative choices of synchronization, we let the user control the upper bound on the cost.
- We first present an exponential algorithm that infers a maximally permissive program for a given language of guards. Next, we define a greedy algorithm that infers, under certain conditions, a maximally permissive program for the given language of guards. Both algorithms minimize synchronization cost.
- We implemented a prototype of our approach and applied it to several programs, including classical ones such as dining philosophers and asynchronous counters.

Next, we use a simple example to illustrate the challenges that our goal presents, and show how they are addressed in our approach.

1.2 A Simple Motivating Example

Fig. 1 is a simple program consisting of three operations `op1`, `op2`, and `op3`, that are executed concurrently by the client program (the `main` procedure). The interleavings for this example are shown in Fig. 2. In this example, the global state consists of the program counter of each of the three threads, and the value of the shared variables `x,y,z`. We denote the global state using a tuple $\langle pc_1, pc_2, pc_3, x, y, z \rangle$ where pc_1,pc_2, pc_3 are program counters and x, y, z are the values of the corresponding shared variables. For this program, we would like to guarantee that the global invariant $y \neq 2 \vee z \neq 1$ is maintained. Unfortunately, while most interleavings indeed satisfy this specification, the interleaving `x=z+1;z=y+1;y=x+1` leads to its violation. In the figure, we use nodes with red dotted boundaries to denote states in which the invariant is violated.

```
op1 { 1: x = z + 1 }        main:
op2 { 2: y = x + 1 }          int x = 0, y = 0, z = 0;
op3 { 3: z = y + 1 }          op1 || op2 || op3
```

Fig. 1. An example program with three threads.

Implementability Our goal in the example is to construct a new maximally permissive program in which the invalid interleaving above is not allowed. Generally, to eliminate invalid traces, we consider the (possibly infinite) set of program traces represented using a transition system, and compute a subset of the transitions in the transition system for which all resulting traces are guaranteed to be accepting. However, since our goal is to construct a program, it is not sufficient to find a valid transition system, we need to find one that is expressible as a program in the provided programming language. Similar *implementability* challenges occur in other synthesis settings, e.g., synthesis of reactive modules [12].

Cost vs. Permissiveness The ability to avoid a specific transition depends on the amount of information that can be obtained atomically from the global state and reflected in a CCR guard. Atomically reading the entire program state is often too costly. Reducing the cost of synchronization is achieved by restricting the language of guards. When the language of guards is restricted, the information available for a guard might not be sufficient to uniquely identify a single transition. This *limited observability* induces a natural equivalence between transitions. Informally, we define two transitions to be equivalent when they execute the same statement, and their source states cannot be distinguished by the language of guards. Under limited observability, the addition of a guard to a statement in order to eliminate a transition t results in the elimination of *all transitions that are equivalent to* t.

Fig. 3(a) shows a valid version of the example of Fig. 1 using CCRs with guards where the bound on the cost allows the solution to observe the entire program state. The synchronization in this program was automatically inferred by our tool. In this program, the guard $(x \neq 1 \vee y \neq 0 \vee z \neq 0)$ (directly) eliminates *only* the transition $\langle e, 2, 3, 1, 0, 0\rangle \xrightarrow{z=y+1} \langle e, 2, e, 1, 0, 1\rangle$ which would have inevitably led to an error state. Note that in this example, allowing the guards to observe the values of all shared variables leads to the maximally permissive result of only eliminating invalid interleavings.

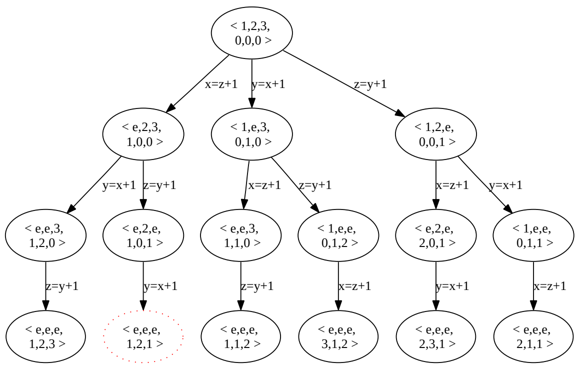

Fig. 2. Transition system for the example program of Fig. 1.(Self-loops on exit states are omitted.)

```
op1 { 1: x = z + 1 }
op2 { 2: y = x + 1 }
op3 { 3: (x≠1 ∨ y≠0 ∨ z≠0) ->
          z = y + 1 }
```

(a)

```
op1 { 1: (x≠0 ∨ z≠0) ->
          x = z + 1 }
op2 { 2: y = x + 1 }
op3 { 3: (x≠1 ∨ z≠0) ->
          z = y + 1 }
```

(b)

Fig. 3. Example program with synchronization, observing (a) all shared variables , (b) only `x,z`.

However, suppose that our solution is restricted to use CCR guards whose cost is limited to only observing the values of variables `x,z` (and not the entire program state). Under such limited observability, the states $\langle e, 2, 3, 1, 0, 0\rangle$, $\langle e, e, 3, 1, 2, 0\rangle$, and $\langle e, e, 3, 1, 1, 0\rangle$ cannot be distinguished by any guard. Therefore, the guard $(x \neq 1 \vee z \neq 0)$ added to the statement `z=y+1` to eliminate the bad transition $\langle e, 2, 3, 1, 0, 0\rangle \xrightarrow{z=y+1} \langle e, 2, e, 1, 0, 1\rangle$ has the side effect of eliminating the two other equivalent transitions. This triggers further elimination of transitions from state $\langle 1, e, 3, 0, 1, 0\rangle$ of statement `x=z+1`. Fig. 3(b) shows a valid solution of the example of Fig. 1 inferred by our tool.

Sometimes it is possible to simplify the guards of a solution *without* affecting the set of allowed interleavings. For example, in Fig. 3(a), we can use only variables `x` and `y` in the guard of `z=y+1`. Such optimizations are further discussed in Section 4.1.

The key point to take away from this example is the connection between synchronization cost and permissibility. Restricting the cost of synchronization by limiting observability may lead to eliminating valid interleavings that cannot be distinguished from invalid ones. For instance, the solution in Fig. 3(b) permits a subset of the traces allowed by the solution in Fig. 3(a) because it is not allowed to observe variable `y`.

In the rest of the paper we describe our approach in more detail. Due to space restrictions, our description is somewhat informal. Additional formal details, examples, proofs, and discussion of related work are available in [16].

2 Preliminaries

Transition System A transition system ts is a tuple $\langle \Sigma, T, \mathit{Init} \rangle$ where Σ is a set of states, $T \subseteq \Sigma \times \Sigma$ is a set of transitions between states, and $\mathit{Init} \subseteq \Sigma$ are the initial states. For a transition $t \in T$, we use $src(t)$ to denote the source state of t, and $dst(t)$ to denote its destination state.

For a transition system ts, we use the following notations. We use $s \rightsquigarrow_{ts} s'$ to denote that there exists a path in ts starting in state s and ending in state s'. Formally, the relation $\rightsquigarrow_{ts}$ is the reflexive transitive closure of the successor relation defined by T. A stuck state is a state that does not have any successors in ts. The set of stuck states is denoted by Stuck_{ts}. A doomed state is a state from which all paths end in stuck states. The set of doomed states is denoted by Doomed_{ts}. We say that a state $s \in \Sigma$ is reachable when there exists a path to s from some initial state. The set of all reachable states of ts is denoted by Reach_{ts}. A transition system ts is valid, denoted by $\mathit{valid}(ts)$, if and only if no doomed state is reachable. For a transition system ts, a trace is a (possibly infinite) sequence of transitions $t_0, t_1, \ldots$ such that $src(t_0) \in \mathit{Init}$ and for every $i \geq 0$, $t_i \in T$ and $dst(t_i) = src(t_{i+1})$. We use $[\![ts]\!]$ to denote the set of traces of a transition system ts. A trace is valid if it does not contain any doomed state.

Conditional Critical Regions (CCRs) The conditional critical region (CCR) construct, originally introduced by Hoare [7], allows the programmer to specify what operations have to be executed atomically and under what condition. A CCR has the form: $guard \rightarrow stmt$ where $guard$ is a boolean expression and $stmt$ is a statement (including a sequential composition of statements) that have to be executed atomically. The guard is evaluated atomically and if *true*, the statements are executed atomically. Otherwise, the thread blocks until the guard evaluates to *true*.

Program Syntax For the purpose of this paper, we consider a program that consists of a set of (named) operations, $\mathit{Op} \overset{\text{def}}{=} \{op_1, \ldots, op_n\}$, executed in parallel by different threads. An operation is a code fragment defined using a simple, flat, programming language with assignment, conditional and unconditional goto, sequential composition, and CCRs. The language does not contain parallel composition, allocation of threads, nested CCRs, and invocation of operations.

If not stated otherwise, each basic statement is in a separate CCR, guarded by $true$, and the guard is omitted. The user may define CCRs in which the atomic statement consists of a sequence of statements, and not a single basic statement. We assume that every statement participates in (exactly one) CCR.

We use *Var* to denote the set of (shared) program variables, which can be referenced by any operation. To simplify the exposition, we do not use local variables. There is nothing in our approach that prevents us from using local variables, but having local variables makes the formal definitions cumbersome. We assume that all program variables have integer values, initialized to 0.

Program Semantics Let P be a program with variables *Var*. A program state s is a pair $\langle pc_s, val_s\rangle$ where $pc_s\colon \{1,\ldots,n\} \rightarrow \mathit{Int}$ maps a thread identifier to the program counter of the corresponding thread, ranging over program locations in the operation executed by the thread, and $val_s\colon \mathit{Var} \rightarrow \mathit{Int}$ is a valuation of the variables. We use Σ_P to denote the set of all program states. The set of initial program states is denoted by $\mathit{Init}_P \subseteq \Sigma_P$. The value of a program expression e in a state $s \in \Sigma_P$ is denoted by $[\![e]\!](s)$. It is computed using standard evaluation rules for program expressions.

We define a transition system for a program P to be $\langle \Sigma_P, T_P, \mathit{Init}_P\rangle$ where a transition $(s, s') \in T_P$ is labeled by a program location l and a thread identifier tid. A transition (s, s') labeled with l and tid is in T_P if (i) the program counter of the thread tid in state s is at program location l, (ii) the guard of the CCR at program location l is satisfied in s, and (iii) execution of the statement corresponding to CCR at l in program state s by thread tid results in state s'. In addition, we guarantee that states at the program exit are not stuck by adding the corresponding self-loop transitions to ts. For a transition $t \in T_P$, we use $\mathit{lbl}(t)$, $tid(t)$, and $ccr(t)$ to denote the corresponding program location, thread id, and the (unique) CCR at program location $\mathit{lbl}(t)$, respectively.

The semantics of a program P, denoted by $[\![P]\!]$, is the (prefix closed) set of traces of the corresponding transition system $\langle \Sigma_P, T_P, \mathit{Init}_P\rangle$.

Specification The user can specify a global invariant S, which describes a set of states. An invariant can refer to program variables and to the program counter of each thread (e.g., to model local assertions). Our approach can be extended to handle any temporal safety specifications, expressed as a property automaton, by computing the synchronous product of program's transition system and the property automaton [3].

We define $\langle \Sigma_P, T_{P,S}, \mathit{Init}_P\rangle$ to be a transition system for a program P and global invariant S, where $T_{P,S} \subseteq T_P$ is defined by removing from T_P all transitions in which the source state does not satisfy S: $T_{P,S} = \{t \in T_P | src(t) \text{ satisfies } S\}$. This effectively means that in the transition system for P and S, all states which do not satisfy S become stuck states — states with no outgoing transitions. If a stuck state is reachable, the transition system for P and S is not valid.

A program P is valid with respect to S if and only if the corresponding transition system $\langle \Sigma_P, T_{P,S}, \mathit{Init}_P\rangle$ is valid. This notion of validity includes both safety properties defined by the global invariant S and a progress guarantee that the program does not get stuck, in any execution.

3 Maximally-Permissive Programs

Given an input program P and a specification S, we modify P by adding synchronization such that the modified program satisfies the specification S. Conceptually, we take the following steps: (i) construct the transition system ts of P and S; (ii) remove a minimal set of transitions from ts such that the resulting transition system ts' is valid with respect to S; (iii) implement ts' as a program, by adding synchronization code to P.

In this work, we rely on standard techniques to construct the transition system of P, e.g., [8], and focus on steps (ii) and (iii).

3.1 Removing Transitions under Limited Observability

By limiting the cost of synchronization code, we induce limited observability. Hence, not every transition system obtained by removing a bunch of transitions from ts can be implemented as a program with the same traces by adding synchronization code to P.

To remove a transition t, and implement the result as a program, the input program P is modified by strengthening the guard of $ccr(t)$, preventing its execution from the source state $src(t)$. When the state $src(t)$ can be uniquely characterized by an expression in the language of guards *LG*, we can use its characterization to strengthen the guard of $ccr(t)$ without affecting transitions other than t. Our ability to uniquely characterize a state $src(t)$ depends on *LG*. Usually, due to limited observability, we may not be able to uniquely characterize $src(t)$. In such cases, the removal of the transition t may remove other transitions executing the same statement, because they have the same guard. We say that two transitions are *equivalent* when the language of guards is not expressive enough to remove one of the transition without removing the other one. We now provide a formal definition of the transition equivalence under limited observability.

Observational Equivalence First, we define equivalence relation on states with respect to *LG*. Two states are equivalent with respect to *LG*, when there is no guard in *LG* that can be used to distinguish them. Formally, for all $s, s' \in \Sigma_P$,

$$s \approx_{LG} s' \text{ if and only if for all } g \in LG. [\![g]\!](s) = [\![g]\!](s') \tag{1}$$

We now define equivalence relation on transitions with respect to *LG*. Two transitions t and t' are equivalent when they execute the same statement and their source states are indistinguishable by *LG*. Formally, for all $t, t' \in T_{P,S}$,

$$t \approx_{LG} t' \text{ if and only if } lbl(t) = lbl(t') \text{ and } src(t) \approx_{LG} src(t') \tag{2}$$

We use $[t]_{LG}$ to denote the equivalence class of t with respect to $\approx_{LG}$. For a set of transitions $E \subseteq T_{P,S}$, we use $[E]_{LG}$ to denote $\cup_{t\in E}[t]_{LG}$.

Characterizing Observable States We define a characterization function to respect the equivalence relation $\approx_{LG}$. Let χ be a function that takes as input a state $s \in \Sigma_P$ and returns a guard in *LG*. We say that χ characterizes the states observable by *LG*, when for all $s, s' \in \Sigma_P$,

$$[\![\chi(s)]\!](s') = true \text{ if and only if } s \approx_{LG} s' \tag{3}$$

Our method is applicable to any guard languages for which a characterization function is defined. Usually, it is easy to define a characterization function, e.g., by enumerating the values of observable variables in the state.

Example 1. Consider the program of Fig. 1 and its transition system in Fig. 2. Let *LG* be boolean combinations of predicates of the form $var == c$, where var is one of the program variables $\{$`x, z`$\}$, and c is a constant. Under *LG*, many of the states in Fig. 2 are equivalent. For example, the states $s_1 = \langle e, 2, 3, 1, 0, 0\rangle$, $s_2 = \langle e, e, 3, 1, 2, 0\rangle$, and $s_3 = \langle e, e, 3, 1, 1, 0\rangle$ are equivalent as they cannot be distinguished by *LG*. Consequently, the transitions corresponding to the statement `z=y+1` outgoing from s_1, s_2, and s_3 are equivalent. When the characterization function is defined by enumerating the values of observable variables in the state, $\chi(s_1) = \chi(s_2) = \chi(s_3) = (x == 1) \wedge (z == 0)$.

3.2 Implementability

We can use χ to define a guard in *LG* that removes a transition $t \in T_{P,S}$, and all the transitions in its equivalence class $[t]_{LG}$, but does not affect any other transitions.

Lemma 1. *For all $t, t' \in T_{P,S}$ such that $lbl(t) = lbl(t')$, $t' \approx_{LG} t$ if and only if $[\![\chi(src(t))]\!](src(t')) = true$.*

A transition system ts is implementable from P and *LG* when there exists a program P' obtained from P by introducing guards from *LG* such that the set of traces of ts and P' are the same. The following theorem relates implementability to observational equivalence. Intuitively, if we remove an equivalence class of transitions from an implementable transition system, the result is an implementable transition system.

Theorem 1 (Implementability). *For every $R \subseteq T_{P,S}$, the transition system ts defined by $\langle \Sigma_P, T_{P,S} \setminus [R]_{LG}, Init_P \rangle$ is implementable from P and LG:*
(1) There exists a program P' such that $[\![P']\!] = [\![ts]\!]$.
(2) P' can obtained from P by introducing guards from LG.

Given P and $[R]_{LG}$, for some $R \subseteq T_{P,S}$, the simple algorithm `implement` from Fig. 4 computes such P'. It relies on Lemma 1 to guarantee that only transitions from $[R]_{LG}$ are removed. The algorithm constructs P' from P by strengthening the guards of CCRs that correspond to transitions in $[R]_{LG}$. For a transition t, we use the notation $P'[l : \neg\chi(src(t)) \wedge guard \rightarrow stmt]$ for the program obtained from P' by strengthening the guard of $ccr(t)$ to be $\neg\chi(src(t)) \wedge guard$. This change is sufficient (by Lemma 1) to remove the transition t itself and all its equivalence class $[t]_{LG}$, but only them.

```
implement(P:Program,R:Transitions):Program {
  $P' = P$
  foreach $t \in R$
    let $ccr(t)$ be $l : guard \rightarrow stmt$ in
    $P' = P'[l : \neg\chi(src(t)) \wedge guard \rightarrow stmt]$
  return $P'$
}
```

Fig. 4. The procedure `implement`.

3.3 Maximally Permissive Programs

We now define the natural notion of a maximally-permissive program for a given language of guards. We note that maximal permissiveness arises in many other settings (e.g., [10, 13]).

Definition 1 (Maximally-Permissive). *Consider a program P and a language of guards LG. Let P' be a program obtained from P by introducing guards from LG. P' is maximally-permissive with respect to LG if and only if P' is valid and for every program P'' obtained from P by introducing guards from LG, if $[\![P']\!] \subset [\![P'']\!]$, then P'' is not valid.*

We use *MP*(P, LG) to denote the set of all maximally-permissive programs that can be obtained from P by introducing guards from *LG*. Note that the programs in *MP*(P, LG) have identical or incomparable sets of traces, i.e., for every pair $P, P' \in$ *MP*(P, LG), $[\![P]\!] \not\subset [\![P']\!]$. When we cannot eliminate all invalid interleavings (that end in stuck states) only by introducing guards, *MP*(P, LG) is empty.

In the rest of this section, we show that every maximally-permissive program can be implemented by removing edges from the transition system of P. We present two algorithms for computing maximally permissive programs with respect to the language of guards *LG*. The language *LG* is required in all of the algorithms. To avoid clutter we do not pass it as an explicit parameter.

3.4 EXHAUSTIVE Algorithm

Theorem 1 allows us to implement any transition system defined by removing a set of transitions $[R]_{LG}$ from the transition system that corresponds to the original program P. We are interested in valid transition systems. Therefore, we restrict our attention to sets of transitions that yield *valid and implementable* transition systems. Rather than considering all subsets of transitions as possible candidates for removal, we define the set of *bad transitions*, and only consider transitions from this set as candidates for removal.

We define a bad transition as a transition that lies on an invalid trace. More formally, given a transition system $\langle \Sigma, T, \mathit{Init} \rangle$ we say that a transition $t \in T$ is a *bad transition* when $i \rightsquigarrow_{ts} src(t), dst(t) \rightsquigarrow_{ts} d$, such that $i \in \mathit{Init}, d \in \mathit{Doomed}_{ts}$. Using this definition, we would like to construct an algorithm that computes a maximally permissive program, but only considers *bad transitions* as candidates for removal.

Side effects Implementability restrictions require that when we remove a transition t we also remove all other equivalent transitions $[t]_{LG}$. As a result, the removal of a bad transition might *introduce* additional bad transitions.

Definition 2. *We say that a removal of a transition t has a* side effect *when* $|[t]_{LG}| > 1$. *When the removal of a transition t does not have a side-effect, we say that it is* side-effect free.

Example 2. Consider the example program of Fig. 1 and its transition system in Fig. 2. Assume that the algorithm has chosen to remove the bad transition $\langle e, 2, 3, 1, 0, 0 \rangle \xrightarrow{z=y+1} \langle e, 2, e, 1, 0, 1 \rangle$, denote it t. The statement executed by this transition is `3:` $true \rightarrow$ `z=y+1`. Under observability limited to variables `x, z`, this removal has the *side effect* of removing the (equivalent) transitions from $\langle e, e, 3, 1, 1, 0 \rangle$ and $\langle e, e, 3, 1, 2, 0 \rangle$. Since there are no other outgoing transitions from these states, the removal of t makes these states doomed, thus adding bad transitions.

Because the removal of a bad transition can introduce additional bad transitions (by introducing doomed and stuck states), an algorithm based on selecting bad transitions has to remove transitions gradually, and recompute the set of bad transitions after every step. This leads to the following algorithm.

Fig. 5 shows the EXHAUSTIVE algorithm for inferring synchronization. The algorithm takes a program as input and constructs a valid program by iteratively eliminating bad transitions. The algorithm maintains a set R of transitions to be removed. Initially, this set is empty. On every iteration of the algorithm, we construct a transition system ts by removing the transitions in R from the transition system of the input program P. If the resulting transition system is valid, the algorithm uses the procedure `implement` to return a modified version of P that avoids all transitions in R. If the transition system ts is not valid, the algorithm computes a set of bad transitions by using the procedure

```
EXHAUSTIVE(P : Program) : Program {
 1:  $R = \emptyset$
 2:  while (true) {
 3:     $ts = \langle \Sigma_P, T_{P,S} \setminus R, \mathit{Init}_P \rangle$
 4:     if valid(ts) return implement(P,R)
 5:     $B =$ bad-transitions($ts$)
 6:     if $B = \emptyset$ abort "cannot find valid synchronization"
 7:     select a transition $t \in B$
 8:     $R = R \cup [t]_{LG}$
     }
}
bad-transitions(ts : TransSys) : Set of Transitions {
   let $ts$ be $\langle \Sigma, T, \mathit{Init} \rangle$ in
   return $\{t \in T \mid i \rightsquigarrow_{ts} src(t), dst(t) \rightsquigarrow_{ts} d, i \in \mathit{Init}, d \in \mathit{Doomed}_{ts}\}$
}
```

Fig. 5. EXHAUSTIVE algorithm.

bad-transitions(ts). If the set is empty, it means that the transition system is not valid, but there are no more bad transitions to be removed (in this algorithm, it means that no bad transitions remain in ts and there exists a stuck state in *Init*). If the set B of bad transitions is not empty, the algorithm non-deterministically chooses one of the transitions in B as the transition to be removed. To guarantee that a program that avoids transitions in R is implementable, when we add a bad transition t to R, we add to R all transitions in its equivalence class $[t]_{LG}$.

Theorem 2 (Correctness of EXHAUSTIVE). *A run of the* EXHAUSTIVE *algorithm terminates with either a valid program or abort.*

Example 3. This example demonstrates how the algorithm is applied to the program of Fig. 1 and its transition system in Fig. 2. The first step in the algorithm is to check whether $ts = \langle \Sigma_P, T_{P,S} \setminus R, \mathit{Init}_P \rangle$ is valid. Since at this point $R = \emptyset$, the transition system is the one of Fig. 2 which is invalid (there is a trace reaching the stuck state $\langle e, e, e, 1, 2, 1 \rangle$). The algorithm now computes the set B, and lets assume that it chooses to remove the bad transition $t = \langle e, 2, 3, 1, 0, 0 \rangle \xrightarrow{z=y+1} \langle e, 2, e, 1, 0, 1 \rangle$. The statement executed by this transition is the statement `3:` $true \rightarrow$ `z=y+1`. Under full observability, $\chi(src(t)) = (x == 1 \wedge y == 0 \wedge z == 0)$. Using this formula, the algorithm creates a new program P' in which the statement has the guard $\neg\chi(src(t))$, that is,
`3:` $(x \neq 1 \vee y \neq 0 \vee z \neq 0) \rightarrow$ `z=y+1`.

Next, we show how to use the EXHAUSTIVE algorithm to compute all maximally permissive programs for a given input program, specification and language of guards. The idea is to implement the non-deterministic choice of a transition $t \in B$ in line 7 using backtracking. As a result, we obtain different sets of transitions to remove, where each set yields a valid program. (It is different from enumerating all possible subsets of bad transitions of the original program, because of side effects.) The following lemma shows that all maximally permissive programs can be obtained this way.

Lemma 2. *For every maximally permissive program $P' \in MP(P, LG)$, there exists a run of the* EXHAUSTIVE *algorithm that returns P'' such that $[\![P']\!] = [\![P'']\!]$.*

Let PS denote the set of (valid) programs obtained from all possible runs of EXHAUSTIVE, for different choices of $t \in B$ in line 7. To compare permissiveness of programs $P_1, P_2 \in PS$, we look at the corresponding sets of removed transitions $R_1, R_2 \subseteq T_{P,S}$, computed by the EXHAUSTIVE algorithm, where $P_i = \texttt{implement}(P, R_i)$, for $i = 1, 2$. If $R_1 \subset R_2$, then the transition system obtained by removing R_1 has more traces (is more permissive) than the transition system obtained by removing R_2. Formally, let *RS* be the set of sets of removed transitions that correspond to the programs in PS. We define the operation $\min(RS)$ that chooses from RS the minimal sets of transitions that guarantee a valid transition system:

$$\min(RS) \overset{\text{def}}{=} \{R \in RS \mid \forall R' \in RS . R' \not\subset R\} \tag{4}$$

This allows us to generate all maximally permissive programs:

Theorem 3. *For every maximally permissive program $P' \in MP(P, LG)$, there exists $R \in min(RS)$ such that $[\![P']\!] = [\![\texttt{implement}(P, R)]\!]$. For every $R \in min(RS)$, $\texttt{implement}(P, R) \in MP(P, LG)$.*

Complexity A single run of EXHAUSTIVE is polynomial in the size of the (original) transition system. The size of RS is exponential in the transition system. Computing $min(RS)$ is polynomial in the size of RS. Therefore, computing $MP(P, LG)$ is exponential in the size of the transition system.

3.5 GREEDY Algorithm

The EXHAUSTIVE algorithm of Fig. 5 is choosing transitions for removal from the set bad-transitions(ts). This set may also contain transitions from one doomed state to another. Removal of a transition between doomed states is redundant, as such a transition will become unreachable (and therefore transitively removed) when transitions into dominating doomed states are removed. We can further leverage the structure of the transition system and avoid removal of a transition between doomed states by having the algorithm pick transitions from the cut between non-doomed and doomed states.

The GREEDY algorithm is a modification of the EXHAUSTIVE algorithm such that instead of using bad-transitions(ts), it uses the following procedure cut-transitions(ts).

```
cut-transitions(ts : TransSys) : Transitions {
  let $ts$ be $\langle \Sigma, T, Init \rangle$ in
  return $\{t \in T \mid i \rightsquigarrow_{ts} src(t), i \in Init, src(t) \notin Doomed_{ts},\ dst(t) \in Doomed_{ts}\}$
}
```

Example 4. Consider the program of Fig. 1. Assume *LG* is as earlier boolean combinations of equality to constants, and is limited to only observing variables `x` and `z`. The starting point of the algorithm is the transition system of Fig. 2. In the first step, the only transition in the cut is the transition $t = \langle e, 2, 3, 1, 0, 0\rangle \xrightarrow{z=y+1} \langle e, 2, e, 1, 0, 1\rangle$, and so the algorithm chooses to eliminate this transition. This results in the addition of the guard $(x \neq 1 \vee z \neq 0)$ to the statement `z=y+1`, and has the side-effect of removing the transitions from $\langle e, e, 3, 1, 1, 0\rangle$ and $\langle e, e, 3, 1, 2, 0\rangle$, which now become doomed states. In the second step, the algorithm chooses to eliminate the transition $\langle 1, e, 3, 0, 1, 0\rangle \xrightarrow{x=z+1} \langle e, e, 3, 1, 1, 0\rangle$. This adds the guard $(x \neq 0 \vee z \neq 0)$ to the statement `x=z+1`, which has the side effect of removing the transition $\langle 1, 2, 3, 0, 0, 0\rangle \xrightarrow{x=z+1}$ $\langle e, 2, 3, 1, 0, 0\rangle$. The resulting program is shown in Fig. 3(b).

Theorem 4 (Correctness of GREEDY). *A run of the* GREEDY *algorithm terminates with either a valid program or abort.*

In GREEDY, the non-deterministic choice of a transition t at line 7 of Fig. 5 returns a cut transition. In contrast to EXHAUSTIVE, where the non-deterministic choice is implemented using backtracking, in GREEDY we implemented it as an arbitrary choice, because any choice returns a reasonable (in fact, locally optimal) solution, while enumerating all possibilities does not guarantee maximal permissiveness. Finding a maximally-permissive solution is exponential in the size of the transition system in the worst-case (using EXHAUSTIVE with backtracking), while GREEDY is polynomial. GREEDY computes a maximally permissive solution when the removal of transitions has no side-effects:

Theorem 5. *If a run of* GREEDY *has no side-effects then it computes a maximally permissive program for P and LG or aborts. If it aborts, then $MP(P, LG) = \emptyset$.*

Note that the theorem only requires that transitions removed during the run of GREEDY to be side-effect free. Recall that under full observability, there cannot be any side-effects, but GREEDY does not require full observability. That is, even under limited observability, it is possible that a run of GREEDY has no side-effects, in which case, it produces a maximally permissive program. However, in cases where limited observability causes side-effects, there are no guarantees: GREEDY may fail or succeed in finding a maximally permissive solution. The following example demonstrates that GREEDY fails to find a maximally permissive program when EXHAUSTIVE manages to find it.

Example 5. Consider the program of Fig. 1, and its transition system in Fig. 2. For this program, when the guard language is limited to only allow the observability of the variable `z`, the result of GREEDY is a program that admits no traces. However, the EXHAUSTIVE algorithm does find a solution with this guard language. The solution found by EXHAUSTIVE is the addition of a guard $z \neq 0$ to the statements `x=z+1` and `z=y+1`.

In most examples we considered, even when GREEDY encountered side-effects, it was always able to find the best solution. Characterizing more accurately when GREEDY guarantees maximal permissiveness is an interesting subject of future work.

3.6 Challenges in Inferring Synchronization under Abstraction

The algorithms presented in this paper operate on a finite transition system. To handle infinite-state systems, we use finite-state abstraction. Given a program P and a specification S, we first compute an abstract transition system for it (see, e.g., [4]), and then apply EXHAUSTIVE or GREEDY to it. If the algorithm does not abort, then the resulting program is guaranteed to satisfy S. However, under abstraction, we cannot guarantee that the resulting program does not reach a stuck state. That is, we might generate guards that make a thread block indefinitely. The reason for this limitation is that under abstraction we might lose the information that a state becomes stuck.

We can conservatively eliminate abstract states that potentially become stuck, losing the ability to guarantee that the result is maximally-permissive. In many cases the conservative approach does not manage to find even a single valid program and aborts. Another approach is to refine an abstract transition system when a state becomes potentially stuck. In the case that the concrete transition system has a finite bisimulation quotient, our algorithm terminates and produces a valid program (or abort). Yet another approach is to use an abstraction that record information about stuck states. There are abstractions that can record some progress properties, but their precision for detecting stuck states has not been evaluated. This is a challenging problem, but it is beyond the scope of this paper.

4 Prototype Implementation

We have implemented the GREEDY algorithm in a prototype tool based on the SPIN model-checker [8]. The tool takes as input a program P, which uses CCRs, a specification S and a set of variables $\mathit{Obs} \subseteq \mathit{Var}$ that guards may refer to. The set of variables Obs is used to determine an upper bound on the synchronization cost. The tool then automatically infers correct synchronization with minimal cost, using the cost function from Section 4.1.

We used the tool on several small but instructive examples, described in [16], including dining philosophers and asynchronous counters. In all of the examples we start with a program that is initially incorrect and does not use any synchronization (our tool also works on input programs that already contain guards). In all examples, out tool successfully inferred guards that achieve maximal permissiveness.

4.1 Reducing Synchronization Cost

The algorithms presented so far infer correct (and maximally permissive) guards whose cost is less than a user-specified upper bound, however, the guards they produce are not guaranteed to have the least synchronization cost for this level of permissiveness. Sometimes, it may be possible to reduce the cost of these guards while maintaining correctness and maximal permissiveness. We now demonstrate how this is done for a specific cost model.

Cost as the Number of Shared Accesses Depending on the environment and the underlying architecture (e.g. cache costs), there may be different cost models for comparing the synchronization cost of two guard expressions. Here, we consider one intuitive cost model: we compare the number of distinct shared variables accessed in each guard. This is a natural measure reflecting the atomic observations about the shared state.

Formally, given a program P, we denote the number of distinct variables accessed by the CCR guard in location l of P by $nga(P, l)$. Given a program P, and a specification S, we say that $P_1 \in \mathit{VP}(P, S)$ has *lower cost* than a program $P_2 \in \mathit{VP}(P, S)$ if for every location l of P, $nga(P_1, l) \leq nga(P_2, l)$.

The language of guards is restricted to boolean combinations of equalities between a variable in the user-provided set Obs and an (integer) constant. We denote this language of guards by $EQ(\mathit{Obs})$ and define a characterization function χ as follows:

$$\chi_{\mathit{Obs}}(s) \overset{\text{def}}{=} \bigwedge_{v \in \mathit{Obs}, [\![v]\!](s)=c} v = c$$

It is easy to see that χ_{Obs} is well defined and characterizes the states observable by the language defined above. The characterization function χ_{Obs} can be extended naturally to apply to sets of states. Given a set of states $S \subseteq \Sigma$, $\chi_{Obs}(S) = \bigvee_{s \in S} \chi_{Obs}(s)$.

The simple version of `implement` shown in Fig. 4 uses χ which finds a guard in the language, but does not attempt to minimize its cost. Synchronization derived using simple version of `implement` always has the same high cost: for each label l, $nga(P, l) = |Obs|$. Our tool uses an improved version of `implement`, shown in Fig. 6, which results in a program with the same permissiveness as for the simple version of `implement`, but has minimal cost. The main idea is to replace χ_{Obs} with a "separator" formula, as we briefly describe next (see [16] for details).

Separator A separator is a guard in *LG* that distinguishes between two sets of states. Given $S_1, S_2 \subseteq \Sigma$, separator g satisfies (i) for all $s_1 \in S_1$, $[\![g]\!](s_1) = true$, and (ii) for all $s_2 \in S_2$, $[\![g]\!](s_2) = false$.

There may be multiple separators in *LG*, with different costs: for example $\chi_{Obs}(S_1)$ is a separator. However, the cost of this separator may be higher than necessary, because it does not take into account S_2. The algorithm in Fig. 6 computes a separator in the language $EQ(Obs)$ with the minimal number of variables. It enumerates subsets V of *Obs* of increasing size until it finds one that can distinguish between S_1 and S_2, and builds a separator formula using χ_V.

The variables in V cannot distinguish between two states s and s' when the values of all these variables are identical in s and s'. Technically, we use $s \downarrow V$ to denote the projection of the state s onto the set of variables $V \subseteq Obs$. For a set $S \subseteq \Sigma$, we use $S \downarrow V$ to denote $\{s \downarrow V \mid s \in S\}$. A set of variables V can distinguish between sets of states S_1 and S_2, if their projections onto V are disjoint.

```
implement(P:Program,R:Transitions) {
 $P' = P$
 $ts = \langle \Sigma_P, T_{P,S} \setminus R, Init_P \rangle$
 $L = \{lbl(t) \mid t \in R\}$
 foreach $l \in L$
   $BS = \{src(t) \in Reach_{ts} \mid lbl(t) = l, t \in R\}$
   $GS = \{src(t) \in Reach_{ts} \mid lbl(t) = l\}$
   $sep$ = SEPARATOR($BS, GS$)
   let $ccr(l)$ be $guard \rightarrow stmt$ in
   $P' = P'[l : \neg sep \wedge guard \rightarrow stmt]$
 return $P'$
}
SEPARATOR($S_1$, $S_2$ : Set of States) {
 foreach $k = 1, \ldots, |Obs|$
   foreach $V \subseteq Obs$ such that $|V| = k$
     if $(S_1 \downarrow V) \cap (S_2 \downarrow V) = \emptyset$
       return $\chi_V(S_1)$;
 abort "cannot find separator"
}
```

Fig. 6. `implement` with separator.

Example 6. Let $Var = \{x, y, z\}$. Let $S_1 = \{\langle 1, 1, 1 \rangle\}$ and $S_2 = \{\langle 1, 1, 2 \rangle, \langle 1, 2, 3 \rangle\}$. Suppose that $Obs = \{x, z\}$. Then, a possible separator for S_1 and S_2 is $\chi_{Obs}(\langle 1, 1, 1 \rangle) \overset{\text{def}}{=} x = 1 \wedge z = 1$, which performs two shared accesses. Another separator for S_1 and S_2 is $z = 1$, and it only accesses a single variable. The algorithm in Fig. 6 returns the latter.

5 Related Work

Early work by Emerson and Clarke [2] uses temporal specifications to generate a synchronization skeleton. The generated programs assume full observability of the program state. This has been later extended by Attie and Emerson to synthesize programs with finer grained atomic sections [1]. Early work by Manna and Wolper [11] synthesizes CSP programs. In contrast, we synthesize programs for shared memory. These approaches have no notion of optimality, and no notion of synchronization cost. Our approach allows us to phrase the question of synchronization cost and optimality relative to a given language of guards. We also assume that the computation performed by the program is provided, and the goal of the synthesis algorithm is to add the required synchronization that guarantees that the specification is satisfied. Pnueli and Rosner [12] consider the problem of synthesizing a reactive module based on an LTL specification. They discuss the problem of *implementability* in this setting, and define necessary and sufficient conditions for the implementability of a given specification.

The work of Joshi et. al. [10] discusses a method for proving that a given program P is maximally concurrent (permissive) with respect to a specification S. This requires a manual phase where the input program P is translated to another equivalent program P' and maximal concurrency is then manually proved on P'. In contrast, we recognize that maximal concurrency is only one component of a more general problem that involves other important dimensions such as synchronization cost. We study how both of these two dimensions are connected and provide algorithms that take into account both dimensions when inferring synchronization.

Acknowledgements: We thank Barbara Jobstmann, and the anonymous referees.

References

1. P.C. Attie and E.A. Emerson. Synthesis of concurrent systems for an atomic read/atomic write model of computation. In *PODC '96*, pages 111–120. ACM, 1996.
2. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, 1982.
3. E.M. Clarke, Jr., O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.
4. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, The Netherlands, December 1996.
5. Brinch Hansen. Edison - a multiprocessor language. *Software - Practice and Experience*, 11(4):325–361, 1981.
6. T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03*, pages 388–402. ACM, 2003.
7. C. A. R. Hoare. Towards a theory of parallel programming. In *The origin of concurrent programming: from semaphores to remote procedure calls*, pages 231–244. 2002.
8. G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
9. B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Conference on Computer Aided Verification (CAV)*, pages 226–238, 2005. LNCS 3576.
10. R. Joshi and J. Misra. Toward a theory of maximally concurrent programs (shortened version). In *PODC '00*, pages 319–328, New York, NY, USA, 2000. ACM.
11. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, 1984.
12. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL '89*, pages 179–190, New York, NY, USA, 1989. ACM.
13. P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.
14. V.A. Saraswat, V. Sarkar, and C. von Praun. X10: concurrent programming for modern architectures. In *PPoPP '07*, pages 271–271, New York, NY, USA, 2007. ACM.
15. H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.
16. M. Vechev, E. Yahav, and G. Yorsh. Inferring synchronization under limited observability. Technical report, IBM, 2008. http://www.research.ibm.com/paraglide/.