

Inferring Synchronization under Limited Observability

Martin Vechev, Eran Yahav, and Greta Yorsh

IBM Research

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 [37], 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 $VP(P, S)$ 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 $VP(P, S)$ can be compared:

- **Permissiveness:** Given two programs $P_1, P_2 \in VP(P, S)$, 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 $P_1, P_2 \in VP(P, S)$, 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. 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 lower 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 her acceptable cost.

There are various synchronization mechanisms made 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 [16]. 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 [13], as a language extension of Java via software transactional memory [14], and recently in the high-performance parallel language X10 [30]. One of the advantages of CCRs over other lower-level operational primitives such as locks and condition variables is their concise and declarative nature.

One of the key challenges in using CCRs is to find 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. This cost is typically dictated by the number of shared variables accessed in the guard, because the guard is evaluated atomically. Hence, one way to reduce the cost is by restricting the code to observe only a subset of these variables. There is a trade-off between the above components. For example, limiting the cost of the guard may reduce the permissiveness. Moreover, manually dealing with 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 $VP(P, S, LG)$ the set of programs that satisfy S and are obtained from P by adding guards from LG . It is possible that no program $P' \in VP(P, S, LG)$ 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: $P' \in VP(P, S, LG)$ is maximally-permissive with respect to LG if there is no program in $VP(P, S, LG)$ 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 $P' \in VP(P, S, LG)$, 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 [19] and controller synthesis [27]. 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, the 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. 3. 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 [27].

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. By reducing the cost of synchronization we restrict the languages 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. 2(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. The transition system for the program of Fig. 2 is shown in Fig. 4 (a). Removed transitions and states are shown as greyed-out. 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.

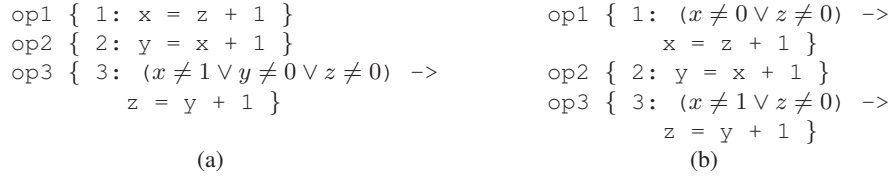


Fig. 2. Operations of the example program with synchronization using observability of (a) all shared variables, and (b) variables x, z .

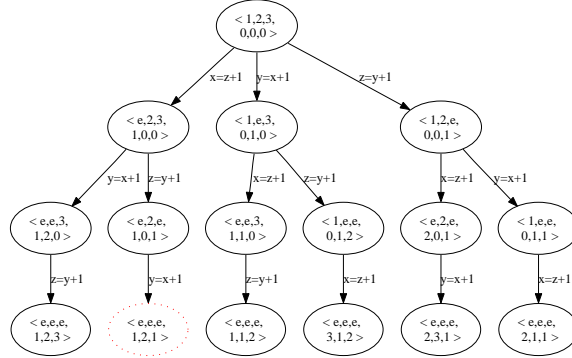


Fig. 3. Transition system for the example program of Fig. 1

Fig. 2(b) shows a valid version of the example of Fig. 1 inferred by our tool, that uses CCR guards whose cost is limited to only observe the values of the variables x, z (not the entire program state). The transition system for the program of Fig. 2 is shown in Fig. 4 (b). Under limited observability, observing only x and z , 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.

The key point is that by restricting the cost on synchronization, we may be forced to use a cheap and coarse synchronization, with limited observability, which cannot distinguish valid interleavings from invalid ones. In such cases, elimination of invalid interleavings leads to unavoidable elimination of the valid ones. For example, the program in Fig. 2(b) permits a subset of the traces permitted by the program in Fig. 2(a). More examples of this connection are provided in Appendix B.

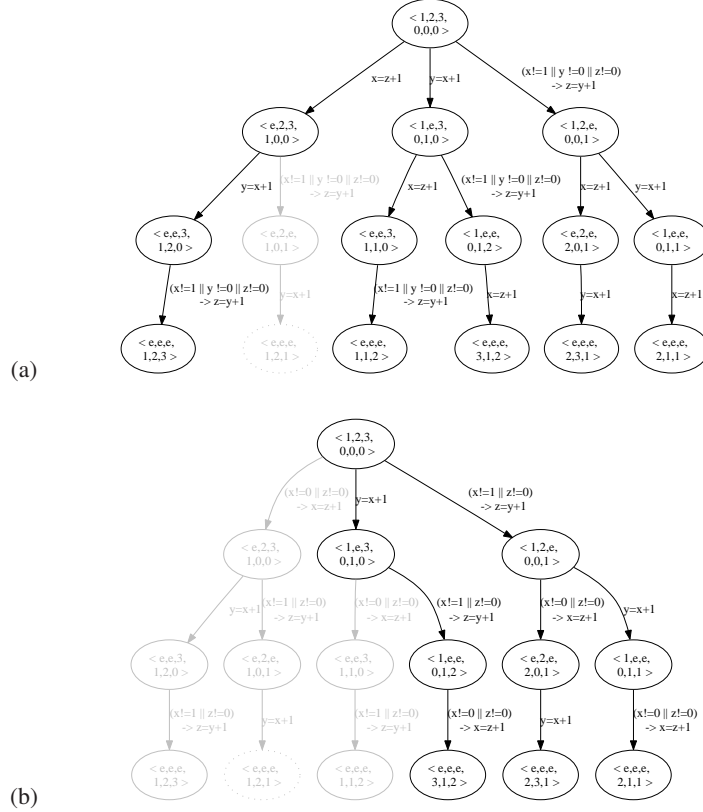


Fig. 4. Different choices of observable state in program guards and the different sets of interleavings they permit: (a) observing all shared variables (Fig. 2), (b) only observe the value of variables x, z (Fig. 2).

Outline The rest of this paper is organized as follows. In Section 2 we provide necessary preliminaries. Section 3 defines two algorithms for computing maximally permissive programs under limited observability. In Section 4 we define an example synchronization cost and show how to minimize it. In Section 5, we discuss the challenges of inferring synchronization under abstraction. In Section 6 we briefly discuss our prototype implementation. Finally, in Section 7 we discuss related work.

The appendix contains additional material: Appendix A describes an algorithm for minimizing the cost function defined in Section 4, Appendix B provides details on our example programs, and Appendix C provides proofs for all theorems stated in Section 3.

2 Preliminaries

Transition System A transition system ts is a tuple $\langle \Sigma, T, Init \rangle$ where Σ is a set of states, $T \subseteq \Sigma \times \Sigma$ is a set of transitions between states, and $Init \subseteq \Sigma$ 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 (see Fig. 5). 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 $valid(ts)$, 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, \dots such that $src(t_0) \in Init$ and for every $i \geq 0$, $t_i \in T$ and $dst(t_i) = src(t_{i+1})$. We use $\llbracket ts \rrbracket$ to denote the set of traces of a transition system ts . A trace is valid if it does not contain any doomed state.

| | |
|------------------------------|--|
| $s \rightsquigarrow_{ts} s'$ | $\stackrel{\text{def}}{=} RTC(T)(s, s')$ |
| $successors(s)$ | $\stackrel{\text{def}}{=} \{s' \mid (s, s') \in T\}$ |
| $Stuck_{ts}$ | $\stackrel{\text{def}}{=} \{s \mid successors(s) = \emptyset\}$ |
| $Doomed_{ts}$ | $\stackrel{\text{def}}{=} LFP(f)(Stuck_{ts})$ |
| f | $\stackrel{\text{def}}{=} \lambda X. X \cup \{s \mid successors(s) \subseteq X\}$ |
| $Reach_{ts}$ | $\stackrel{\text{def}}{=} \{s \in \Sigma \mid i \rightsquigarrow_{ts} s, i \in Init\}$ |
| $valid(ts)$ | $\stackrel{\text{def}}{=} Doomed_{ts} \cap Reach_{ts} = \emptyset$ |

Fig. 5. Definitions for a transition system $ts = \langle \Sigma, T, Init \rangle$.

Conditional Critical Regions (CCRs) The conditional critical region (CCR) construct, originally introduced by Hoare [16], is an elegant construct that 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, $Op \stackrel{\text{def}}{=} \{op_1, \dots, op_n\}$, and a client. 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.

A client initializes the variables and invokes k threads in parallel, for some k : $op_{i_1} \parallel \dots \parallel op_{i_k}$, where $1 \leq i_{tid} \leq n, 1 \leq tid \leq k$. Several threads may execute the same operation. We use $op(tid)$ to denote the operation invoked by a thread whose identifier is tid , for $1 \leq tid \leq k$.

Program Semantics Let P be a program with variables Var . A program state s is a pair $\langle val_s, pc_s \rangle$ where $val_s: Var \rightarrow Int$ is a valuation of the variables and $pc_s: \{1, \dots, k\} \rightarrow Int$ is the program counter of each thread, which ranges over program locations in the operation executed by the thread. We use Σ_P to denote the set of all program states. The set of initial program states is denoted by $Init_P \subseteq \Sigma_P$. The value of a program expression e in a state $s \in \Sigma_P$ is denoted by $\llbracket e \rrbracket(s)$. It is computed using standard evaluation rules for program expressions.

We define a transition system for a program P to be $\langle \Sigma_P, T_P, Init_P \rangle$ 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 $lbl(t)$, and $tid(t)$ to denote the corresponding program location and thread id. We use $ccr(t)$ to denote the (unique) CCR at program location $lbl(t)$.

The semantics of a program P , denoted by $\llbracket P \rrbracket$, is the (prefix closed) set of traces of the corresponding transition system $\langle \Sigma_P, T_P, Init_P \rangle$.

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 [8].

We define a transition system for a program P and global invariant S to be $\langle \Sigma_P, T_{P,S}, Init_P \rangle$ 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 : $T_{P,S} = \{t \in T_P \mid src(t) \text{ satisfies } S\}$. 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 successor transitions. In such cases, 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 $\langle \Sigma_P, T_{P,S}, Init_P \rangle$ 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.

Because we deal with non-deterministic programs in which a state may have more than one transition, a scheduler can pick one of these transitions and executes it. We

assume strong fairness of the scheduler, i.e., if a transition is enabled infinitely often, it is taken infinitely often.

In the next sections, we often refer to a transition system obtained from the transition system of P and S by removing some transitions: $ts = \langle \Sigma_P, T, Init_P \rangle$, where $T \subseteq T_{P,S}$.

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., [17], and focus on steps (ii) and (iii).

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 by adding synchronization code to P .

3.1 Removing Transitions under Limited Observability

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 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 the available language of guards. Usually, due to limited observability, we are not able to uniquely characterize $src(t)$.

When the language of guards cannot uniquely characterize $src(t)$, the removal of the transition t cannot be implemented without removing 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 $s, s' \in \Sigma_P$,

$$s \approx_{LG} s' \text{ if and only if for all } g \in LG. \llbracket g \rrbracket(s) = \llbracket g \rrbracket(s') \quad (1)$$

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 $t, t' \in T_{P,S}$,

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

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 $s, s' \in \Sigma_P$,

$$\llbracket \chi(s) \rrbracket(s') = \text{true} \text{ if and only if } s \approx_{LG} s' \quad (3)$$

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. 3. 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. 3 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 $\llbracket \chi(src(t)) \rrbracket(src(t'))$.*

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 $\llbracket P' \rrbracket = \llbracket ts \rrbracket$.*
- (2) *P' can be obtained from P by introducing guards from LG .*

Given P and $[R]_{LG}$, for some $R \subseteq T_{P,S}$, the simple algorithm `implement` shown in Fig. 6 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 $P'[l : \neg\chi(src(t)) \wedge guard \rightarrow stmt]$ to denote the program obtained from P' by strengthening the guard of $ccr(t)$ to be $\neg\chi(src(t)) \wedge guard$. Note that by Lemma 1 this change is sufficient to remove the transition t itself and all its equivalence class $[t]_{LG}$, but only them.

```

implement(P: Program, R : Set of Transitions) : Program {
  P' = P
  foreach t ∈ R
    let ccr(t) be l : guard → stmt in
      P' = P'[l : ¬χ(src(t)) ∧ guard → stmt]
  return P'
}

```

Fig. 6. 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., [21, 29]).

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 $\llbracket P' \rrbracket \subset \llbracket P'' \rrbracket$, 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 $P_1, P_2 \in MP(P, LG)$, $\llbracket P_1 \rrbracket \not\subset \llbracket P_2 \rrbracket$. 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 and S . 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 $\langle \Sigma, T, Init \rangle$ 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 $i \in Init, d \in Doomed_{ts}$. 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.

```

EXHAUSTIVE(P : Program) : Program {
1:  R = ∅
2:  while (true) {
3:    ts = ⟨ΣP, TP,S \ R, InitP⟩
4:    if valid(ts) return implement(P,R)
5:    B = bad-transitions(ts)
6:    if B = ∅ abort "cannot find valid synchronization"
7:    select a transition t ∈ B
8:    R = R ∪ [t]LG
  }
}
bad-transitions(ts : TransSys) : Set of Transitions {
  let ts be ⟨Σ, T, Init⟩ in
  return {t ∈ T | i ~_{ts} src(t), dst(t) ~_{ts} d, i ∈ Init, d ∈ Doomedts}
}

```

Fig. 7. EXHAUSTIVE algorithm.

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. 3. Assume that the algorithm has chosen 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 observability limited to the 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.

The EXHAUSTIVE algorithm Fig. 7 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 `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 transitions in ts remain and all states in $Init$ are stuck). 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. 3. The first step in the algorithm is to check whether $ts = \langle \Sigma_P, T_{P,S} \setminus R, Init_P \rangle$ is valid. Since at this point $R = \emptyset$, the transition system is the one of Fig. 3 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 $\neg sep$, 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.

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 $\llbracket P' \rrbracket = \llbracket P'' \rrbracket$.*

Let PS denote the set of (valid) programs obtained from all possible runs of EXHAUSTIVE, for different choices of $t \in B$. To compare permissiveness of two 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, such that $P_i = implement(P, R_i)$, 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 from 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:

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

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 $\llbracket P' \rrbracket = \llbracket implement(P, R) \rrbracket$. For every $R \in min(RS)$, $implement(P, R) \in MP(P, LG)$.*

Complexity A single run of EXHAUSTIVE is polynomial in the size of the 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. 7 is choosing transitions for removal from the set $\text{bad-transitions}(ts)$. 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 $\text{bad-transitions}(ts)$, it uses the following procedure $\text{cut-transitions}(ts)$.

```
cut-transitions(ts : TransSys) : Transitions {
  let ts be ⟨Σ, T, Init⟩ in
  return {t ∈ Tts | i ↘ts src(t), i ∈ Init, src(t) ∉ Doomedts, dst(t) ∈ Doomedts}
}
```

Example 4. Consider the program of Fig. 1. Assume the language of guards is as earlier boolean combinations of equality to constants, and is limited to only observing the values of variables x and z . The starting point of the algorithm is the transition system of Fig. 3. 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. The statement executed by t is $3: \text{true} \rightarrow z=y+1$. Therefore, this step 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 transition from $\langle e, e, 3, 1, 1, 0 \rangle$ and $\langle e, e, 3, 1, 2, 0 \rangle$, which now become doomed states. The transition system after this step is shown in Fig. 8. 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 transition system after this step is shown in Fig. 4 (b), and only permits valid traces. The resulting program is shown in Fig. 2.

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

When the removal of transitions has no side-effects, we can show that GREEDY can compute a maximally permissive program for a given input program, specification and language of guards.

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. It does not require full observability. That is, even under limited observability, the algorithm can produce maximally permissive results. 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.

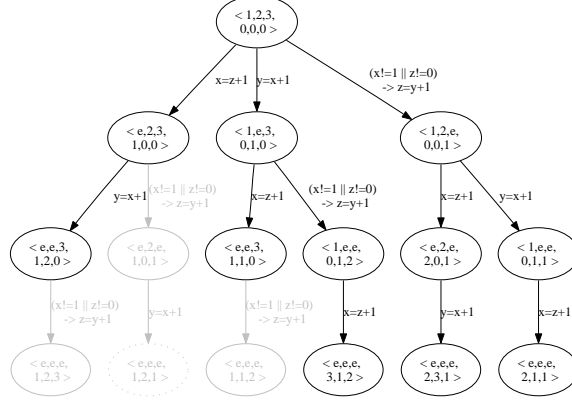


Fig. 8. First step of the GREEDY algorithm for the example of Fig. 1 with observability limited to variables x, z .

Example 5. Consider the program of Fig. 1, and its transition system in Fig. 3. 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 practical examples we considered, GREEDY was always able to find the best solution, even with side-effects. Characterizing more accurately when GREEDY guarantees maximal permissiveness is a subject of future work.

4 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 that level of permissibility. Therefore, it may be possible to further reduce the cost of the 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. specific cache costs), there may be many 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 $P_1 \in VP(P, S)$ has *lower cost* than a program $P_2 \in VP(P, S)$ 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 $EQ(Obs)$ and define a characterization function χ . The function

characterizes the observable part of a state by conjoining the values of all observable program variables:

$$\chi_{Obs}(s) \stackrel{\text{def}}{=} \bigwedge_{v \in Obs, \llbracket v \rrbracket(s)=c} v = c$$

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. 6 uses a characterization function 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`, which results in a program with the same permissiveness as for the simple version of `implement`, but has minimal cost. This version of `implement` is based on the notion of a separator, explained in Appendix A.

5 Challenges in Inferring Synchronization under Abstraction

The algorithms presented in this paper operate on a finite transition system. To apply our technique to infinite-state systems, we can use finite-state abstraction.

Given a program P and a specification S , we first compute an abstract transition system for it (see, e.g., [9]). Intuitively, we partition the set of concrete states Σ into a finite number of equivalence classes, A , the states of the abstract transition system. There is a transition (a, a') in the abstract transition system if there exists a transition t in the concrete transition system for P and S such that $src(t)$ and $dst(t)$ are in the partitions a and a' , respectively.

```

op1 {
  1: x = x + 1
  2: y = y + 1
  3: goto 1
}
op2 {
  4: x = x - 1
  5: y = y - 1
  6: goto 4
}
main {
  x = 0, y = 0;
  op1 || op2
}

```

Fig. 9. Example program with an infinite state space.

Example 6. Fig. 9 shows a simple example program that has an infinite state space. For this example program, we would like to make sure that the invariant $I = (pc_2 = 6) \Rightarrow even(x + y)$. To apply our algorithm for this program, we employ a simple abstraction that abstracts the value of variables to their parity. Fig. 10 shows the synchronization inferred for this program under the parity abstraction.

We can apply any of the algorithms from Section 3 to an abstract transition system. 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

```

op1 {
  1:  $((\neg \text{even}(x) \vee \neg \text{even}(y)) \wedge ((\text{even}(x) \vee \text{even}(y)))) \rightarrow x = x + 1$ 
  2:  $y = y + 1$ 
  3: goto 1
}
op2 {
  4:  $x = x - 1$ 
  5:  $((\neg \text{even}(x) \vee \neg \text{even}(y)) \wedge ((\text{even}(x) \vee \text{even}(y)))) \rightarrow y = y - 1$ 
  6: goto 4
}

```

Fig. 10. Inferred synchronization for the infinite state program of Fig. 9.

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, e.g., [5], but their precision for properties of stuck states has not been evaluated. This is a challenging problem, but it is beyond the scope of this paper.

6 Prototype Implementation

We have implemented a prototype tool based on the GREEDY algorithm. The tool takes as input a program P , which uses CCRs, a specification S and a set of variables $Obs \subseteq Var$ that guards may refer to. This set of variables is used to determine an upper bound on the synchronization cost. The tool then automatically infers correct synchronization with minimal cost (with cost function defined in Section 4). The implementation is based on the SPIN model-checker [17].

We used the tool on several small but instructive examples. In all of the examples we start with a program that is initially incorrect and does not use any synchronization. We use the tool to automatically infer the maximally permissive synchronization for the given synchronization cost. The examples illustrate the connection between synchronization cost and permissiveness. We describe these examples in Appendix B.

7 Related Work

Synthesis from Temporal Specifications Early work by Emerson and Clarke [7] 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 [2]. Early work by Manna and Wolper [23] 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 is to phrase the question of synchronization cost and optimality as maximal permissiveness under limited observability of the 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. Our work can be viewed as supervisory controller synthesis [29], because we control the program by observing its state and blocking certain interleavings, without modifying the state. Pnueli and Rosner [27] 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.

Program Repair Jobstmann et. al. [19] consider the problem of *program repair* as a game. In their approach, a game is constructed from (a modified version of) the program to be repaired, and an LTL specification of the correctness property. The problem of repair boils down to finding a winning strategy in that game. This approach has been later extended to provide fault localization and fixing [36, 20]. Similarly to us, the work of Janjua et al. [18] presents an algorithm that eliminates incorrect program traces in a concurrent program by inserting blocking calls. However, they do not discuss synchronization cost or its inherent connection to permissiveness. Finally, our algorithms can also be viewed as a special case of inferring a maximally-permissive *memoryless* winning strategy in a game with incomplete information.

Maximal Concurrency The work of Joshi et. al. [21] 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.

Formal Derivation Another related line of research is the systematic derivation of concurrent programs from formal specifications [1, 12]. Typically, such derivations start from a correct sequential or a coarse-grained atomic program and at each step prove that its refinement to a finer-grained atomic program is correct. This approach is typically used to *re-construct* existing algorithms, rather than to study how to create new ones. Usually, these methods are not concerned with maximal concurrency or observability (as they are reconstructing an existing algorithm with already fixed dimensions).

Dynamic Approaches The problem of restricting the program to valid executions can be addressed by monitoring the program at *runtime* and forcing it to avoid executions that violate the specification. However, restricting the executions of a program at runtime requires a recovery mechanism in case the program already performed a step that violates the specification, and/or a predictive mechanism to check whether future steps lead to a violation.

Existing approaches using recovery mechanisms typically require additional user annotations to define a corrective action to be taken when the specification is violated. For example, software transactional memory [31] is a special case of a recovery mechanism in which the user provides atomicity annotations defining atomic sections. The system then requires the absence of read/write conflicts, and if this property is violated, the execution of an atomic section is restarted. Other recent examples include Tolerance

[26] which creates local copies of variables to detect and recover from race conditions, and ISOLATOR [28] which can recover from violations of isolation.

Existing approaches that use predictive mechanisms such as deadlock avoidance methods [10], are generally only successful for limited classes of properties, as the predictive mechanism needs to predict the violation in a state that still permits its avoidance, which is a hard problem.

Search-based Synthesis In previous work [39, 38], we used a semi-automated approach for exploring a space of various concurrent algorithms: concurrent garbage collectors and linearizable data-structures. The work used a search procedure and an abstraction specifically geared towards the safety property required for the specific domain.

In the work of [4], the authors deal with mutual exclusion algorithms. They perform syntactic exploration and discover various interesting algorithms, some of which are better than known solutions under the given space constraints.

In *sketching* [34, 35, 32], the user provides a reference program of the desired implementation and some sketches which partially specifies certain optimized functions. The sketching compiler automatically fills in the missing low-level details to create an optimized implementation. Sketching has been used to synthesize several bitstream program implementing cryptographic ciphers. More recently [33], sketching has been used to generate concurrent data-structures based on a bounded-checking procedure.

In previous work [39], we used a semi-automated approach for exploring a space of concurrent garbage collector algorithms. That work used a limited search procedure and an abstraction specifically geared towards the safety property required for that specific domain. In [38], we used a similar search procedure for deriving linearizable data-structures. The work of [24, 3] on superoptimization finds the shortest instruction sequence to compute a function.

Locks for Atomicity Recently, there have been several works on pessimistic implementation of atomic sections (usually assuming weak atomicity). In the work by McCloskey et. al. [25], a tool called Autolocker is presented. The tool takes as input a program that has been manually annotated with (i) atomic sections and (ii) a mapping between locks and memory locations protected by these locks. Autolocker takes this input and produces a program that implements the atomic sections in (i) with the locks in (ii). Further work by Emmi et. al. [11] proposed a technique to automate part (ii) above. The actual assignment of locations to locks is solved as an optimization problem where the goal is to minimize the total number of locks while still achieving minimum interference between the computed locks. The latest work of Cherem et. al. [6] proposes another alternative to automate requirement (ii) while also computing the actual lock placement in the code. Our work is complementary to these approaches, as our focus is not on optimizing the implementation of CCRs, but on studying how to restrict non-determinism (here it is technically via CCRs) as a function of two fundamental entities: user-provided invariants *and* state observability. A complementary research challenge is taking a program annotated with CCRs (of which *atomic* is a special case), and translating it to a semantically equivalent program that uses lower level synchronization primitives such as locks and condition variables efficiently.

References

1. Jean-Raymond Abrial and Dominique Cansell. Formal construction of a non-blocking concurrent queue algorithm (a case study in atomicity). *J. UCS*, 11(5):744–770, 2005.
2. 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.
3. Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. *SIGOPS Oper. Syst. Rev.*, 40(5):394–403, 2006.
4. Yoah Bar-David and Gadi Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *Proc. of the symp. on Principles of Distributed Computing*, pages 305–305, 2003.
5. J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P.W. O’Hearn. Variance analyses from invariance analyses. In *POPL*, pages 211–224, 2007.
6. Sigmund Chorem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *PLDI’08*, pages 304–315, New York, NY, USA, 2008. ACM.
7. 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.
8. E.M. Clarke, Jr., O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.
9. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, The Netherlands, December 1996.
10. E.W. Dijkstra. Cooperating sequential processes, TR EWD-123. Technical report, 1965.
11. Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL '07*, pages 291–296, New York, NY, USA, 2007. ACM.
12. Lindsay Groves and Robert Colvin. Derivation of a scalable lock-free stack algorithm. *Electron. Notes Theor. Comput. Sci.*, 187:55–74, 2007.
13. Brinch Hansen. Edison - a multiprocessor language. *Software - Practice and Experience*, 11(4):325–361, 1981.
14. T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03*, pages 388–402. ACM, 2003.
15. M.P. Herlihy and J.M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), 1990.
16. 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. Springer-Verlag New York, Inc., New York, NY, USA, 2002.
17. G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
18. Muhammad Umar Janjua and Alan Mycroft. Automatic correction to safety violations in programs. In *Thread Verification*, 2006.
19. 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.
20. B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem. Finding and fixing faults. *Journal of Computer and System Sciences (JCSS)*, 2008. Accepted for publication.
21. 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.
22. L. Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, 1977.
23. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, 1984.
24. Henry Massalin. Superoptimizer: a look at the smallest program. In *ASPLOS-II: Architectural support for programming languages and operating systems*, pages 122–126. IEEE, 1987.
25. Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL '06*, pages 346–358, New York, NY, USA, 2006. ACM.

26. Rahul Nagpaly, Karthik Pattabiraman, Darko Kirovski, and Benjamin Zorn. Tolerace: Tolerating and detecting races. In *STMCS: Second Workshop on Software Tools for Multi-Core Systems*, 2007.
27. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL '89*, pages 179–190, New York, NY, USA, 1989. ACM.
28. Sriram Rajamani, G. Ramalingam, Venkatesh-Prasad Ranganath, and Kapil Vaswani. Controlling non-determinism for semantic guarantees. In *Exploiting Concurrency Efficiently and Correctly – (EC)2*, 2008.
29. 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.
30. 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.
31. Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95*, pages 204–213, New York, NY, USA, 1995. ACM.
32. Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.
33. Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching concurrent data structures. In *PLDI '08*, pages 136–148, New York, NY, USA, 2008. ACM.
34. Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294. ACM, 2005.
35. Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.
36. S. Staber, B. Jobstmann, and R. Bloem. Finding and fixing faults. In *Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 35–49, 2005. LNCS 3725.
37. H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.
38. Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI '08*, pages 125–135, New York, NY, USA, 2008. ACM.
39. Martin T. Vechev, Eran Yahav, David F. Bacon, and Noam Rinetzk. Cgexplorer: a semi-automated search procedure for provably correct concurrent collectors. In *PLDI '07*, pages 456–467, New York, NY, USA, 2007. ACM.

A Separator

A simple version of `implement` is shown in Fig. 6. It uses a characterization function 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 , $nga(P, l) = |Obs|$.

In this section, we present another version of `implement`, which results in a program with the same set of traces as for the simple version of `implement`, but has minimal cost. The new version of `implement` is shown in Fig. 11 and is based on the notion of a separator which we describe next.

Separator Intuitively, a separator is a guard in LG that distinguishes between two sets of states.

Definition 3 (Separator). *Let LG be a language of guards. Let $S_1, S_2 \subseteq \Sigma$ be sets of states. A separator for S_1 and S_2 in LG is a guard $g \in LG$ such that for all $s_1 \in S_1$, $\llbracket g \rrbracket(s_1) = true$ and for all $s_2 \in S_2$, $\llbracket g \rrbracket(s_2) = false$.*

A separator for S_1 and S_2 does not exist in LG if and only if there are states $s_1 \in S_1$ and $s_2 \in S_2$ such that $s_1 \approx_{LG} s_2$.

```

implement(P: Program, R : Set of Transitions) : Program {
  P' = P
  ts = ⟨ΣP, TP,S \ R, InitP⟩
  L = {lbl(t) | t ∈ R}
  foreach l ∈ L
    BS = {src(t) | t ∈ R ∧ lbl(t) = l}
    GS = {src(t) ∈ Reachts | lbl(t) = l}
    sep = SEPARATOR(BS, GS)
    let ccr(l) be guard → stmt in
      P' = P'[l : ¬sep ∧ guard → stmt]
  return P'
}

```

Fig. 11. The procedure `implement` using separator to optimize cost.

Lemma 3. *If a separator for S_1 and S_2 in LG exists, then it can be defined using the characterization function $\chi: \Sigma \rightarrow LG$:*

$$\chi_{Obs}(S_1) \text{ is a separator for } S_1 \text{ and } S_2$$

However, the cost of this separator may be higher than necessary, because it does not take into account S_2 .

Given S_1 and S_2 , there may be multiple separators in LG , with different costs. Note that all separators for S_1 and S_2 are logically equivalent over the set $S_1 \cup S_2$, but $\chi_{Obs}(S_1)$ is the strongest separator in Σ . By choosing a weaker separator, we can reduce its cost, as follows.

Given the observable variables Obs , we can define separator using χ_{Obs} , as in Lemma 3. However, the cost of this separator is high because it enumerates all variables in Obs , even if these variables are not required to distinguish between S_1 and S_2 , e.g., evaluate to the same values in all states $S_1 \cup S_2$.

The algorithm in Fig. 12 computes a separator with the minimal number of variables. It enumerates subsets 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 .

```

SEPARATOR( $S_1, S_2$  : Set of States) : Guard {
  foreach  $k = 1, \dots, |Obs|$ 
    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. 12. An algorithm for computing a separator for disjoint sets of states $S_1, S_2 \subseteq \Sigma$ with respect to observable variables Obs .

The variables $V \subseteq Obs$ cannot distinguish between 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.

Example 7. 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 $\chi_{Obs}(\langle 1, 1, 1 \rangle) \stackrel{\text{def}}{=} x = 1 \wedge z = 1$, 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. 12 returns the latter.

The correctness of this algorithm relies on the fact that in $EQ(V)$ the observational equivalence can be decided by comparing values of the variables in V only, without evaluating all guards in the language:

Lemma 4. *For all $s, s' \in \Sigma$, $s \approx_{EQ(V)} s'$ if and only if $(s \downarrow V) = (s' \downarrow V)$.*

B Examples

In this section, we use the tool to illustrate the connection between synchronization cost and permissiveness on several small but instructive examples. In these examples, we use the synchronization cost as defined in Section 4. In all of the examples we start with a program that is initially incorrect and does not use any synchronization. We use the tool to automatically infer the maximally permissive synchronization for the given cost (observability).

B.1 Asynchronous Counters

```

counter1 {          counter2 {          main() {
  A: x = x%3 + 1;    B: y = y%2 + 1;    x = 1; y = 1;
  goto A;           goto B;           counter1 || counter2;
}                  }                  }

```

Fig. 13. Asynchronous Counters

| Guard | Full { x,y } | Full/Sep { x,y } | limited { y } | limited { x } |
|-------|----------------------------|----------------------------|-------------------------|-------------------------|
| A | $(x \neq 2 \vee y \neq 1)$ | $(x \neq 2 \vee y \neq 1)$ | $(y \neq 1)$ | $(x \neq 2)$ |
| B | $(x \neq 3 \vee y \neq 2)$ | $(x \neq 3)$ | $(y \neq 2)$ | <i>true</i> |

Fig. 14. Guards computed using GREEDY for the CCRs at locations A and B of the asynchronous counters example, under different settings of observability.

Consider the example program of Fig. 13. This program consists of two threads: each thread continuously performs modulo counting on a variable different than the other thread. The threads operate independently of each other, e.g. use no synchronization. The challenge is to synchronize the two counter threads so that certain shared states are never reached. That is, no external reader accessing the shared state is able to observe such values. In this example, assume that every state must satisfy the following invariant: $\neg(x = 3 \wedge y = 1)$.

Fig. 14 summarizes the results we obtained for this example by running GREEDY with different settings of observability. In the table we show the guards computed for the CCRs at locations A and B of the example program.

Full Observability In this case, the synchronization cost allows the solution to observe all shared variables, that is, full observability. Assuming that both counter threads can observe the shared variables x and y , the GREEDY algorithm produces the following

valid program as a result:

```

counter1 {
  A:  $(x \neq 2 \vee y \neq 1)$  ->
    x = x%3 + 1;
    goto A;
}
counter2 {
  B:  $(x \neq 3 \vee y \neq 2)$  ->
    y = y%2 + 1;
    goto B;
}

```

That is, it produces the guards $(x \neq 2 \vee y \neq 1)$ and $(x \neq 3 \vee y \neq 2)$ for A and B respectively, as shown in Fig. 14. These guards correspond to the maximally permissive solution that can be obtained for this example as they eliminate only the invalid traces.

Applying Separator Applying the separator of Section 4 to this example improves the cost of the solution. That is, it simplifies the guard of the second counter thread while maintaining the same level of permissibility. The result is shown in column *Full/Sep* in Fig. 14.

Limited observability However, let us suppose the solution is only allowed to observe variable y . In this case, the system produces the guards $(y \neq 1)$ and $(y \neq 2)$ for A and B respectively, shown in column *limited y* in Fig. 14. This solution is maximally permissive with respect to the limited observability, but permits a subset of the traces allowed by the solution using full observability. In both guards of this limited observability solution, only a single shared access is performed (to the variable y) and hence application of the separator will not improve the cost.

Now, assume that instead of y , we lower the cost and allow it to observe only variable x . In this case, the system produces the guards $(x \neq 2)$ and *true* for A and B respectively (shown as last column of Fig. 14). In this solution, only the guard for A was strengthened (as compared to the original program). The result is maximally permissive for this restricted cost, but is less permissive than the one computed with full observability.

We have exercised the tool with more complex versions of this example, involving multiple counter and reader threads. However, this simple example is the essential representative and clearly illustrates that limiting synchronization cost limits the permissiveness.

Discussion It is interesting to note that there are various concurrent snapshot algorithms which ensure that readers cannot observe intermediate updates to shared memory by writers (see for example the concurrent reading and writing problem as discussed by Lamport [22]). Many of these algorithms are designed to satisfy general atomicity specifications such as linearizability [15]. However, if such an algorithm is taken and used in a specific setting, it may restrict the allowed concurrency unnecessarily and in unknown ways. For example, in our case, if we are to use an algorithm where each cycle of increments is atomic (e.g. the only observable state is when both x and y are 0), then such an algorithm would be correct but unnecessarily restrictive for our invariant. Note that we can still use our approach to infer synchronization to satisfy general atomicity correctness criteria.

B.2 Dining Philosophers

Our next example is a slight adaptation of dining philosophers, a well known problem in concurrency control, first proposed and solved by Dijkstra [10]. In this setup, there are N philosophers dining on a round table. Each philosopher alternates its state between thinking and eating. The challenge is to design a synchronization protocol which avoids deadlock (no philosopher will ever eat) and starvation (every philosopher will eventually eat). There are many variants and solutions of this problem, all with different levels of concurrency. Assume for example that each philosopher atomically updates its state as a function of its previous state (we have ran examples where that is not the case). Moreover, the philosophers are not synchronized with each other. Initially, when each philosopher starts its operation, its state is set to THINKING.

```

deposit {                withdraw {
  instr_r_x = 1;         instr_r_y = 1;
  l_x = bal;             l_y = bal;
  instr_r_x = 0;         instr_r_y = 0;
  l_x = l_x + 5;         l_y = l_w - 2;
  instr_w_x = 1;         instr_w_y = 1;
  bal = l_x;             bal = l_y;
  instr_w_x = 0;         instr_w_y = 0;
}                        }
main {
  bal = 5;
  instr_r_x = instr_w_x = 0;
  instr_r_y = instr_w_y = 0;

  deposit || withdraw
}

```

Fig. 15. Race correction example

```

philosopher(int i) {
  A: phil[i] = (phil[i]==THINKING) ? phil[i]=EATING : phil[i]=THINKING;
  goto A;
}

```

Our safety criteria is that no two adjacent philosophers can eat at the same time. Assume that we have four philosophers sitting on a table where philosopher 0 is neighbors with 1 and 3, 1 with 0 and 2, 2 with 1 and 3, and 3 with 0 and 2.

Full Observability We start by having the cost of the synchronization code observe all variables. Assume the state of each philosopher is observable by all other philosophers. If we run the system without using the separator, the tool produces guards that are fairly elaborate. However, if separator is used, the cost is reduced (e.g. smaller guards are computed) and we end up with the expected (and maximally permissive) solution. Below is the expected inferred CCR guard for philosopher 0 produced by the tool:

```
(!phil[1] == EATING ^ !phil[3] == EATING) -> ...
```

Limited Observability Let us now suppose that we restrict the synchronization cost and do not allow the solution to observe the state of philosopher 3. When we run the system (with separator), the following maximally permissive solution is produced (we do not show the guards here without separator, but they are more expensive, although similarly permissive).

```

0: (false -> ...)
1: (true -> ...)

```

```

2: (false -> ...)
3: (true -> ...)

```

The above solution is also intuitive. Without being able to observe the state of philosopher 3, the neighbors of 3 (0 and 2) can never make progress, because otherwise they may end up eating together with 3. It is easy to see that by decreasing observability, we have ended up with a correct solution that is less permissive than the one with full observability. Note that we consider this a correct solution because we ignore starvation of philosophers. Even though both solutions allow starvation, with full observability, there are executions where philosophers 0 and 3 can make progress, while when observability is limited, we are forced to err on the safe side and never allow any progress of 0 and 3.

B.3 Race Correction

In this example we show how we used the system to fix data races. A *race condition* occurs if two threads access a shared variable at the same time, and at least one of these accesses is a write.

Our current implementation is limited to invariant specifications. Because race conditions are a temporal safety property, we used additional instrumentation to encode it in our current implementation. In the future, we plan to extend our tool to directly handle temporal safety properties without additional instrumentation.

We consider the commonly used illustrative example of a bank account (see Fig. 15) with two methods accessing a shared variable. For this simple example, we manually introduced instrumentation. We introduce a variable for each shared access and set it to 1 right before the shared access and reset it to 0 right after the shared access. The instrumentation variables are in bold.

Echoing the definition of a race condition above, the state safety invariant is that no two instrumentation variables from different threads that associate to the same location, where at least one of them is a location that is written, are both set to 1. The invariant is $\neg(\mathbf{instr_w_y} = \mathbf{instr_w_x} = 1 \vee \mathbf{instr_w_y} = \mathbf{instr_r_x} = 1 \vee \mathbf{instr_r_y} = \mathbf{instr_w_x} = 1)$. Running the system computes a solution that is race-free and maximally permissive. Below we show only the inferred guards (using the separator):

```

deposit {
  ...
  !(instr_w_y == 1) -> instr_r_x = 1;
  !(instr_r_y == 1  $\vee$  instr_w_y = 1) -> instr_w_x = 1;
  ...
}

withdraw {
  ...
  !(instr_w_x = 1) -> instr_r_y = 1;
  !(instr_r_x = 1  $\vee$  instr_w_x = 1) -> instr_w_y = 1;
  ...
}

```

C Proofs

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

Proof: Let $t, t' \in T_{P,S}$ such that $lbl(t) = lbl(t')$. If $t \approx_{LG} t'$, then by definition of the equivalence relation \approx_{LG} in (2), we get that $src(t) \approx_{LG} src(t')$. Therefore, by (3), we conclude that $\llbracket \chi(src(t)) \rrbracket (src(t')) = \text{true}$. If $t \not\approx_{LG} t'$ then by definition of the equivalence relation \approx_{LG} we get that $src(t) \not\approx_{LG} src(t')$. Therefore, by (3), we conclude that $\llbracket \chi(src(t)) \rrbracket (src(t')) = \text{false}$.

Theorem 1 Let $R \subseteq T_{P,S}$. 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 $\llbracket P' \rrbracket = \llbracket ts \rrbracket$.
- (2) P' can be obtained from P by introducing guards from LG .

Proof: Let $R \subseteq T_{P,S}$ and P' be the result of $\text{implement}(P, [R]_{LG})$ defined in Fig. 6. In this algorithm, the program P' is obtained from P by introducing guards from LG . To show that $\llbracket P' \rrbracket = \llbracket ts \rrbracket$, we show that the transition system of P' is the same as ts . It boils down to showing that $T_{P',S} = T_{P,S} \setminus [R]_{LG}$.

Recall that the algorithm in Fig. 6 modifies the program P by strengthening the guards, i.e., the transitions of P' is a subset of the transitions of P : $T_{P',S} \subseteq T_{P,S}$. In other words, there exists a set R' of transitions such that $T_{P',S} = T_{P,S} \setminus R'$. We show that $R' = [R]_{LG}$.

It is easy to see that $[R]_{LG} \subseteq R'$: for every $t \in [R]_{LG}$, the algorithm in Fig. 6 strengthens the guard of the $ccr(t)$ with $\chi(src(t))$, therefore, t is not enabled in P' , i.e., $t \in R'$.

Consider a transition t' that is in P but not in P' , i.e., $t' \in R'$. We show that $t' \in [R]_{LG}$. The guard of $ccr(t')$ is defined by strengthening the guard of $ccr(t')$ in P , in a way that eliminates t' . That is, there exists $t \in [R]_{LG}$ such that $lbl(t) = lbl(t')$ and $src(t)$ satisfies $\chi(src(t))$. By Lemma 1, $t' \approx_{LG} t$ and thus $t' \in [R]_{LG}$.

Corollary C1 Let $R \subseteq T_{P,S}$ be closed under \approx_{LG} , i.e., $R = [R]_{LG}$.

$$\llbracket \text{implement}(P, R) \rrbracket = \llbracket \langle \Sigma_P, T_{P,S} \setminus R, Init_P \rangle \rrbracket$$

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

Proof: In every iteration, the EXHAUSTIVE algorithm adds at least one transition from $T_{P,S}$ to the set R . Because $T_{P,S}$ is finite, the EXHAUSTIVE algorithm either finds that ts is valid or that B is empty, and terminates. If the transition system ts is valid, then the EXHAUSTIVE algorithm returns a program $\text{implement}(P, R)$, denoted by P' , where R is closed under \approx_{LG} . By Corollary C1, we get that $\llbracket P' \rrbracket = \llbracket ts \rrbracket$, and thus P' is a valid program.

Lemma C2 For every $P' \in MP(P, LG)$, there exists $R \in T_{P,S}$, such that R is closed under \approx_{LG} and $\llbracket P' \rrbracket = \llbracket \langle \Sigma_P, T_{P,S} \setminus R, Init_P \rangle \rrbracket$.

Proof: By Definition 1 of $MP(P, LG)$, P' is obtained from P by introducing guards in LG such that $\llbracket P' \rrbracket \subseteq \llbracket P \rrbracket$ and P' is valid. Since P' is less permissive than P , but has the same statements, it differs from P by having stronger guards, i.e., the transition

system of P' has less transitions than that of P . Therefore, there exists R such that $T_{P',S} = T_{P,S} \setminus R$.

Suppose that R is not closed under \approx_{LG} . Let $t, t' \in T_{P,S}$ such that $t \approx_{LG} t'$, $t \notin R$ and $t' \in R$, i.e., $t \in T_{P',S}$, and $t' \notin T_{P',S}$. Therefore, the guard $g \in LG$ of $ccr(t)$ in P' holds for $src(t)$ but not for $src(t')$, and a contradiction is obtained with $src(t) \approx_{LG} src(t')$.

Definition C3 Let P' be a program obtained from P by introducing guards in LG , and $R \in T_{P,S}$. We say that R is a **clean cut for P'** when all the following conditions hold:

- (a) R is closed under \approx_{LG} ,
- (b) $\llbracket P' \rrbracket = \llbracket \text{implement}(P, R) \rrbracket$,
- (c) for all $t \in R$ there exists $t' \in R$, such that $t \approx_{LG} t'$ and $src(t')$ is reachable in P' .

Lemma C4 Let $R' \in T_{P,S}$ be closed under \approx_{LG} and let P' be the result of $\text{implement}(P, R')$. There exists $R \subseteq R'$ such that R is a clean cut for P' .

Proof: Let $t \in R'$. If for all transitions $t' \in [t]_{LG}$, $src(t')$ is not reachable in P' , then we can remove $([t]_{LG}t)$ from R' without changing the set of traces, i.e., $\llbracket \text{implement}(P, R') \rrbracket = \llbracket \text{implement}(P, R' \setminus [t]_{LG}) \rrbracket$. Therefore, we define R by removing from R' all the equivalence classes that are unreachable in P' and get that $\llbracket \text{implement}(P, R) \rrbracket = \llbracket P' \rrbracket$. By construction, R is closed under \approx_{LG} and $R \subseteq R'$.

Lemma C5 Let R be a clean cut for P' , and let $R'' \subset R$, such that R'' is closed under \approx_{LG} . If $P' \in MP(P, S)$ then the program $\text{implement}(P, R'')$ is invalid.

Proof: First, we show that the program $\text{implement}(P, R'')$ has strictly more traces than P' . By Definition C3 of clean cut, at least one transition in $R \setminus R''$ is reachable in P' , but does not belong to P' , therefore, adding that transition to the transition system means adding a new trace. Since P' is maximally permissive, any program that has strictly more traces is invalid, by Definition 1. In particular $\text{implement}(P, R'')$ is invalid.

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 $\llbracket P' \rrbracket = \llbracket P'' \rrbracket$.

Proof: Let P' be a program in $MP(P, LG)$. We show that there exists a run of EXHAUSTIVE that returns P'' such that $\llbracket P' \rrbracket = \llbracket P'' \rrbracket$.

By Lemma C2, let $R' \subseteq T_{P,S}$ be a set of transitions closed under \approx_{LG} such that $\llbracket P' \rrbracket = \llbracket \langle \Sigma_P, T_{P,S} \setminus R', \text{Init}_P \rangle \rrbracket$. By Corollary C1, we get that P' and $\text{implement}(P, R')$ have the same semantics (sets of traces). By Lemma C4, there exists a set $R \subseteq R'$ such that R is a clean cut for $\text{implement}(P, R')$, and thus $\llbracket P' \rrbracket = \llbracket \text{implement}(P, R) \rrbracket$. It remains to show that there exists a run of EXHAUSTIVE that returns $\text{implement}(P, R)$. That is, we need to show that R can be computed by some choices of bad transitions in EXHAUSTIVE.

If the original program P is valid, then $R = \emptyset$. We assume that R contains m equivalence classes of transitions, for $m > 0$. The proof is by induction on the number of iterations of the algorithm. For $i > 0$, let R_i denote the value of R at the end of iteration i of EXHAUSTIVE; ts_i denote the transition system in iteration i , i.e., $\langle \Sigma_P, T_{P,S} \setminus R_{i-1}, \text{Init}_P \rangle$; B_i denote the set of bad transitions of ts_i , i.e., $B_i = \text{bad-transitions}(ts_i)$; and t_i be the transition chosen in iteration i from B_i . Initially, $R_0 = \emptyset$.

We show by induction that for all $i = 1, \dots, m$, if $R_{i-1} \subseteq R$ then $B_i \cap R \neq \emptyset$. It implies that in each iteration, we can pick a transition t_i such that $t_i \in B_i \cap R$, and add

its equivalence class to R_{i-1} to get $R_i \subseteq R$. Following this order, we get that $R_m = R$ and in the next iteration ts_{m+1} is valid and EXHAUSTIVE returns $\text{implement}(P, R)$ as required.

Intuitively, we can always pick t_i from $B_i \cap R$, because every reachable stuck state in ts_i has a path to it through R (because, if we remove R from ts_i , we get a valid transition system). Thus, R will contain at least one bad transition in ts_i , i.e., $B_i \cap R \neq \emptyset$.

Let $1 \leq k \leq m$. Suppose that for all $i = 1, \dots, k-1$, we pick $t_i \in B_i \cap R$. Note that we get that $R_{i+1} \subset R$ and thus removing R from ts_i we get the same traces as in P' .

We show that EXHAUSTIVE does not return or abort in iteration k . First, we show that ts_k is not valid (**return** in line 4 of EXHAUSTIVE). Recall that R is a cut of P' and $P' \in MP(P, S)$. By Lemma C5, we get that $\text{implement}(P, R_k)$ is not valid because $R_k \subset R$. Since $\llbracket ts_k \rrbracket = \llbracket \text{implement}(P, R_k) \rrbracket$, we get that ts_k is not valid. Second, we show that B_k is not empty (**abort** in line 6 of EXHAUSTIVE). If B_k is empty then it is impossible to remove transitions from ts_k to make doomed states unreachable, therefore removing R from ts_k would not result in a valid program, contradicting the fact that P' is valid.

We show that there exists $t \in B_k \cap R$. If $B_k \cap R$ is empty, then removing R from ts_k does not make any doomed states of ts_k unreachable, i.e., the result is not a valid transition system, contradicting the fact that P' is valid.

Lemma C6 *For every maximally permissive program $P' \in MP(P, LG)$, there exists $R \in \min(RS)$ such that $\llbracket P' \rrbracket = \llbracket \text{implement}(P, R) \rrbracket$.*

Proof: Recall that the set R , defined in the proof of Lemma 2, belongs to RS and satisfies $\llbracket P' \rrbracket = \llbracket \text{implement}(P, R) \rrbracket$. We show that $R \in \min(RS)$.

By definition of $\min(RS)$, we need to show that for all $R'' \in RS$, $R'' \not\subset R$. For the sake of contradiction, suppose that there exists $R'' \in RS$ such that $R'' \subset R$. Since R is a clean cut for P' and $P' \in MP(P, S)$, we can use Lemma C5 to get that $\text{implement}(P, R'')$ is not valid. By definition of RS , $R'' \notin RS$ and a contradiction is obtained.

Lemma C7 *For every $R \in \min(RS)$, $\text{implement}(P, R) \in MP(P, LG)$.*

Proof: Let $R \in \min(RS)$. By definition, $\min(RS) \subseteq RS$, and for all $R' \in RS$, R' is computed by EXHAUSTIVE and induces a valid program $\text{implement}(P, R')$. Thus, in particular, the program $\text{implement}(P, R)$, denoted by P_1 , is valid. We can use Corollary C1, because R is closed under \approx_{LG} , to get that $T_{P_1, S} = T_{P, S} \setminus R$. For the sake of contradiction, suppose that P_1 is not maximally permissive, i.e., there exists a valid program P_2 obtained from P by introducing guards in LG , such that $\llbracket P_1 \rrbracket \subset \llbracket P_2 \rrbracket$. Without loss of generality, we assume that P_2 is maximally-permissive, i.e., $P_2 \in MP(P, S)$.

Using Lemma C6, there exists $R_2 \in \min(RS)$ such that $\llbracket P_2 \rrbracket = \llbracket \text{implement}(P, R_2) \rrbracket$. Note that R_2 is closed under \approx_{LG} . Using Corollary C1, we get that $\llbracket P_2 \rrbracket = \llbracket \langle \Sigma_P, T_{P, S} \setminus R_2, \text{Init}_P \rangle \rrbracket$. It implies that $T_{P_2, S} \supseteq T_{P, S} \setminus R_2$.

The programs P_1 and P_2 differ only in the guards, and not statements, but $\llbracket P_1 \rrbracket \subset \llbracket P_2 \rrbracket$. Therefore, the transitions system of P_2 has strictly more transitions than that of P_1 , i.e., there exists R_1 such that $T_{P_1, S} = T_{P_2, S} \setminus R_1$ and R_1 is not empty.

Combining the above constraints about $T_{P_1, S}$, we get that $T_{P, S} \setminus R \supseteq T_{P, S} \setminus (R_2 \cup R_1)$. Therefore, $R \supseteq R_1 \cup R_2$ and since R_1 is not empty we get the strict inclusion:

$R \supset R_2$. This, together with the facts that $R_2 \in RS$ and $R \in \min(RS)$, leads to contradiction.

Theorem 3

Proof: Follows from Lemma C6 and Lemma C7

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

Proof: Similar to the proof of Theorem 2

Lemma C8 *Suppose that a run of GREEDY is side-effects free and it computes a set R of edges. Let ts denote the transition system of P . For all $t \in R$, $\text{dst}(t) \in \text{Doomed}_{ts}$.*

Proof: Recall that a side-effect free run means that for all $i > 0$, the GREEDY algorithm in iteration i picks a transition t from $\text{cut-transitions}(ts)$ such that $|\llbracket t \rrbracket_{LG}| = 1$. Therefore, in iteration $i + 1$, the set R is the same as in iteration i union with t , and the transition system in $i + 1$ is the same as in i minus t . By removing t from the transition system, reachability of other cut transitions is not affected, because every (simple) path contains at most one cut-transition. Thus, the cut-transitions of $i + 1$ are the same as in i , minus t . It follows that $R = \text{cut-transitions}(ts)$. By definition, the destination of a cut transition is a doomed state, which completes the proof that destinations of all transitions in R are doomed states.

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 $MP(P, LG) = \emptyset$.*

Proof: Suppose that a run of GREEDY without side-effects computes a set R of edges and returns the valid program $\text{implement}(P, R)$, denoted by P' . We show that $P' \in MP(P, LG)$.

Let P'' be a program obtained from P by introducing guards from LG , such that $\llbracket P' \rrbracket \subset \llbracket P'' \rrbracket$. Since P' and P'' differ only in guards, and not statements, and they both obtained from P , the transitions of P'' are a superset of those of P' , and a subset of those of P . Therefore, $T_{P'',S} \setminus T_{P',S} \subset R$. It follows that every trace in $\llbracket P'' \rrbracket \setminus \llbracket P' \rrbracket$ contains at least one transitions from R . By Lemma C8, R is a set of doomed transitions in P . A trace that contains a doomed state is not valid. Therefore, the program P'' , which contains an invalid trace, is not valid.