SATCheck: SAT-Directed Stateless Model Checking for SC and TSO

Brian Demsky
University of California, Irvine, USA
bdemsky@uci.edu

Patrick Lam
University of Waterloo, Canada
patrick.lam@uwaterloo.ca

Abstract

Writing low-level concurrent code is well known to be challenging and error prone. The widespread deployment of multi-core hardware and the shift towards using low-level concurrent data structures has moved the problem into the mainstream. Finding bugs in such code may require finding a specific bug-revealing thread interleaving out of a huge space of parallel executions.

Model-checking is a powerful technique for exhaustively testing code. However, scaling model checking presents a significant challenge.

In this paper we present a new and more scalable technique for model checking concurrent code, based on concrete execution. Our technique observes concrete behaviors, builds a model of these behaviors, encodes the model in SAT, and leverages SAT solver technology to find executions that reveal new behaviors. It then runs the new execution, incorporates the newly observed behavior, and repeats the process until it has explored all reachable behaviors.

We have implemented a prototype of our approach in the SATCheck tool. Our tool supports both the Total Store Ordering (TSO) and Sequentially Consistent (SC) memory models. We evaulate SATCheck by testing several concurrent data structure implementations and comparing its performance to the original DPOR stateless model checking algorithm implemented in CDSChecker, the source DPOR algorithm implemented in Nidhugg, and CheckFence. Our experiments show that SATCheck scales better than previous approaches while at the same time operating on concrete executions.

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3689-5/15/10...$15.00
http://dx.doi.org/10.1145/2814270.2814297

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Relaxed memory model; model checking

1. Introduction

Testing concurrent code can be challenging: exposing bugs often requires driving program execution to a specific interleaving. However, bug-revealing interleavings may account for a vanishingly small proportion of a large interleaving space. Model checking can be an effective way of testing concurrent code, but current model checking approaches are typically best suited for small unit tests. In this work, we present a novel approach that leverages SAT solving technology to explore only the interleavings that expose new concrete behaviors.

Current approaches to model checking concurrent code primarily take one of three approaches:

- **Explicit State:** Early work on model checking concurrent code explicitly modeled program state. This approach is not often used directly on software, as encoding the entire state can be problematic. Furthermore, encoded state can lead to a combinatorial state explosion even when all operations commute.
- **Stateless:** Stateless approaches to model checking eliminate the need to explicitly encode program state and instead explore all interleavings of conflicting (i.e. non-commutative) operations. In the context of stateless model checking, researchers have developed several partial order reduction techniques to reduce exploration of redundant executions [1, 16–18].

  Partial order reduction techniques for stateless model checking reason locally about the commutativity of individual operations and effectively assume that any operation may depend on all previous operations. There remains further potential for improvement in partial order reduction by incorporating reasoning about the program's dependency structure.

To be more precise, consider a program with two threads that both first perform an operation on a concurrent queue and then each perform an independent operation on a concurrent stack. Existing POR approaches will explore every interleaving of the stack operations in the context of every interleaving of the queue operations even though these operations are independent!

- **SAT-Based Approaches:** Researchers have improved on stateless model checking by developing SAT-based approaches to model checking [9, 28]. The key insight is that, by encoding a program as a SAT formula, the model checker can leverage the SAT solver's heuristics to avoid wasting time exploring redundant executions.

  Current SAT-based approaches translate the entire program to a SAT formula. This is non-ideal in that many programs contain arithmetic expressions, which significantly complicate the SAT formula and limit the scalability of this approach. Moreover, the approach is not applicable to code that calls library functions whose source is not available.

  A second downside of existing SAT-based approaches is that they are all or nothing—if the generated SAT equation is too complex for the SAT solver to analyze, the model checker provides the developer with no information.

We introduce a new approach to model checking concurrent code that leverages two key insights from the previous approaches:

- **Reasoning about dependences is necessary for scalability:** Approaches that naïvely run code don't scale because they waste too much effort exploring redundant executions. Scalable approaches must reason about how operations depend on each other to avoid wasting effort on redundant executions.
- **Encoding the entire program in SAT can add significant complexity from the non-concurrent parts of the code:** Approaches that try to encode the entire program into SAT incur complexity from the non-concurrent parts of the computation (e.g., arithmetic on integers). In many cases, concurrent executions of a given test case cannot drive arbitrary values through the program, and thus it is unnecessary to encode how the computation operates on all values. Only values that actually arise in concurrent executions must be encoded.

1.1 Our Approach

This paper presents our novel SAT-based approach to model checking for concurrent code. Unlike previous SAT-based approaches, we use the SAT solver to encode the execution, not the program. Our approach was inspired by concolic testing and previous work on model checking concurrent code [27]. SATCheck leverages dependency information provided in program instrumentation. We have developed a compiler frontend that instruments C code for use with SATCheck. SATCheck accepts as input an instrumented program including stores, loads, atomic Read-Modify-Write (RMW) operations, uninterpreted functions, equality comparisons, structured conditional branches, while loops, and phi functions.

Our approach, like similar approaches in stateless model checking, (DPOR [16, 35], Chess [23], SourceDPOR [2], Optimal DPOR [1], Maximal Causality Reduction [20], etc.) begins by fixing a set of user-provided inputs, and concretely executing the program under test with these inputs. We then use the results of the concrete execution to construct an *event graph* representation of the program's observed behaviors. The event graph captures the observed control flow paths of the program; observed input-output relations for uninterpreted functions; observed memory operations, conditional branches, and loops; and dependences between each of these components. SATCheck then translates the event graph into a SAT formula that captures the observed behaviors of the program. SATCheck next adds clauses to this SAT formula, that, when satisfied, generate an execution that expands SATCheck's knowledge of the program's behavior—either by taking a new (unexplored) direction on a branch, by visiting a novel interleaving, or by learning a new input-output relation for an uninterpreted function. SATCheck uses an off-the-shelf SAT solver to solve the SAT formula. If there is no solution, then no interleaving will yield new behaviors—SATCheck has explored all program behaviors (for the given program input). If there is a solution, SATCheck generates a interleaving from the solution and repeats the process. Once SATCheck has converged, the SAT formula describes all observable behaviors of the program under test for a particular set of inputs to that program.

Our tool is available under an open source license at the following URL:

`http://plrg.eecs.uci.edu/satcheck/`

1.2 Contributions

This paper makes the following contributions:

- **Basic Approach:** It presents a new technique for model checking concurrent code. The approach learns the behaviors of the program through concrete executions and uses a SAT solver to search for executions that reveal new program behaviors.
- **TSO Support:** After developing the algorithm for the sequentially consistent memory model, it presents an extension of the algorithm to Total Store Ordering (TSO).
- **SATCheck Implementation:** It presents SATCheck, a prototype implementation of our model checking technique. Our implementation includes a Clang-based frontend that instruments C programs for use with SATCheck.
- **Evaluation:** It presents an evaluation of the SATCheck implementation on several concurrent data structures. Our evaluation shows that SATCheck scales to much

larger problem sizes and runs more quickly than previous concrete execution-based approaches.

2. Example

```
1   typedef struct lock_t {
2     int lock;
3   } lock;
4
5   lock a;
6
7   void initlock(lock *l) {
8     l->lock = 0;
9   }
10  bool trylock(lock *l) {
11    int val=cas(&l->lock, 0, 1);
12    return val==0;
13  }
14  void unlock(lock *l) {
15    store(&l->lock, 0);
16  }
17  void foo() {
18    if (trylock(&a)) {
19      unlock(&a);
20    }
21  }
```

Figure 1. C spin lock implementation.

We explain how SATCheck works using the simple spin lock example shown in Figure 1. This example implements methods `trylock` and `unlock`, which are called by the driver `foo` method.

Assume a program which creates two threads, both of which call `foo`. SATCheck starts by concretely executing the program. Assume that in the first execution (1) Thread 1 first executes the CAS operation in its `trylock`, then (2) Thread 2 executes the CAS operation in its `trylock`, and finally (3) Thread 1 executes the `store` in `unlock`. After SATCheck observes this execution, it constructs the initial event graph in Figure 2.

2.1 Event Graph Construction

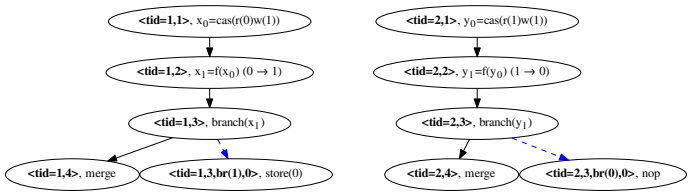

Figure 2. Event graph summarizes first execution (T_1 CAS, T_2 CAS, T_1 store) with unconditionally executed nodes (connected by solid black arrows) and conditionally executed nodes (dashed blue arrows).

Each operation instance in the execution is assigned a unique *execution point* (or EP) tuple, shown in <**bolded angle brackets**> in Figure 2. EP tuples allow SATCheck to match equivalent operation instances between different executions and reflect the nested structure of the programming language. An EP tuple starts with the thread identifier and includes a sequence of counts. SATCheck maintains per-thread EP counters during the program's execution. The execution counter for a thread starts as a 2-tuple consisting of the thread identifier along with a single counter set to 1.

Figure 2 shows that Thread 1's first CAS operation gets EP `<tid=1,1>`. Since this operation runs first, it reads the value 0 and writes the value 1, as indicated by the label `r(0)w(1)`. Also, SATCheck assigns a unique identifier for the output of each event graph node. Thread 1's first CAS operation gets the identifier x_0.

After a normal operation (e.g. load, store, RMW, or function invocation), SATCheck increments the last element of the EP. Thus, the second operation, an uninterpreted function that evaluates `val==0`, gets EP `<tid=1,2>`. The label x_1=`f(`x_0`)` indicates that the function depends on the value x_0 produced by the previous CAS operation and that the identifier x_1 gets the output of the function. SATCheck remembers the input-output relation of the function: the label $0 \rightarrow 1$ indicates that SATCheck has observed that for an input of 0, function `f` produces output 1.

The following branch operation gets EP tuple `<tid=1,3>`. After a branch operation, SATCheck appends the direction of the branch and a new zeroed counter to the EP. Thus, the conditionally executed `store` operation has EP `<tid=1,3,br(1),0>`. The fragment `br(1)` indicates that this operation is only executed when the enclosing conditional branch is taken. We graphically indicate conditionally executed code with a dashed blue arrow. The black arrow shows the next statement to be executed after the code enclosed by the conditional branch finishes.

The event graph also includes information on Thread 2's execution. The explanation is analogous to that for Thread 1.

From this event graph, SATCheck deduces the possibility of behaviors that it has not yet observed: (1) an execution where Thread 1 skips the body of its `if` statement; (2) an execution where Thread 2 executes the body of its `if` statement; and (3) executions where the two uninterpreted functions are evaluated on input values that differ from the current inputs. An execution with a different interleaving will allow SATCheck to evaluate the uninterpreted functions on different inputs.

2.2 Translation to SAT

SATCheck next translates the event graph into a SAT formula that describes the observed behaviors of the program and the execution interleaving. It then adds *goal* clauses that encode a set of constraints that, when satisfied, ensure that the program exhibits a new behavior. New execution interleavings potentially yield new program behaviors that satisfy these goal clauses — the SAT solver searches for such interleavings which drive the program to produce new behaviors.

Like CheckFence [9], SATCheck represents the execution interleaving by assigning a single boolean *interleaving variable* to every pair of memory operations from different

threads. If the boolean variable is true, the first memory operation appears earlier in the interleaving. If it is false, the second memory operation appears earlier in the interleaving. (Unlike CheckFence, our approach operates on concrete executions.) For our running example, SATCheck generates the SAT variable $v_{\texttt{<tid=1,1>},\texttt{<tid=2,1>}}$, which is true when operation `<tid=1,1>` occurs before operation `<tid=2,1>`. SATCheck also generates the variable $v_{\texttt{<tid=1,3,br(1),0>},\texttt{<tid=2,1>}}$ for the store in Thread 1. Finally, SATCheck generates constraints between these variables to enforce transitivity properties (to ensure that the interleaving is an ordering).

SATCheck uses a potential value set analysis on the event graph to determine the values that variables (including addresses) may take, in the absence of novel behaviors. Novel behaviors create new, previously unseen values. Potential value sets do not capture all possible values, only the values a variable may have if the execution does not encounter new program behaviors. The potential value set analysis does not need to account for new values; they will be generated in later iterations by concrete program iterations. In particular, SATCheck assumes that uninterpreted functions do not generate new values.

On our running example, the analysis tells us that (1) the CAS and store operations only access one address, (2) the values written to that address are 0 and 1, and (3) the input values to the memory operations are fixed. Note that new values are created by generating executions that create these values and SATCheck generates goal clauses to ensure that it produces all possible values. Future runs of the potential value set analysis would then include those values.

Because the example uses fixed addresses for its memory operations, SATCheck does not need to allocate any SAT variables for addresses. Since the memory operations only store two possible values ($\{0, 1\}$), SATCheck encodes them using a single SAT variable to index into the set.

We also allocate value variables for memory operations, and allocate two variables per CAS operation (one for the value read, $r_{\texttt{<tid=1,1>}}$ and $r_{\texttt{<tid=2,1>}}$, and one for the value stored, $w_{\texttt{<tid=1,1>}}$ and $w_{\texttt{<tid=2,1>}}$) and one SAT variable $w_{\texttt{<tid=1,3,br(1),0>}}$ for the store, representing the value it stores. For each load operation, we encode the store it reads-from using SAT variables. The CAS operation `<tid=1,1>` can read-from one of two stores: the CAS operation `<tid=2,1>` or the initialization store. Thus we can use a SAT variable $\ell_{\texttt{<tid=1,1>}}$ to represent which of these two stores it reads-from. If variable ℓ indicates that a load reads-from a given store, then (1) their values must match, (2) the store must be ordered before the load, and (3) there cannot be a conflicting store to the same address ordered between them. SATCheck generates constraints to capture these properties.

The uninterpreted functions take as input the value returned by the CAS operation (0 or 1 from the potential value set). Consider the uninterpreted function $\texttt{<tid=1,2>x}_1 = \texttt{f}(\texttt{x}_0)(0 \rightarrow 1)$. We model this function's output as either the previously observed value 1 or a special value to represent an as-yet unobserved output. One SAT variable can represent these two cases.

This graph contains two branches. Both branches have taken one direction each; the remaining directions are unexplored. Thus we represent the state of the branch with two values: either the explored direction, or a new direction. We can therefore allocate one SAT variable per branch. From the current execution, we know that if the uninterpreted function returns 1, then the `if` body will be executed. Thus we can build an implication constraint from the SAT variable for the output of the uninterpreted function to the SAT variable for the branch direction.

The effect of store `<tid=1,3,br(1),0> store`(0) must be predicated on taking the branch. Thus the branch SAT variable appears in the constraints for loads to ensure that if some executed load reads from this store, then the store was in fact executed.

Forcing Novel Executions The key novelty of our approach is that we model sets of observed concrete executions of the program (including dependencies between computed values) and iterate to find new, interesting executions. This greatly reduces the state space that must be explored. To support this approach, SATCheck generates a clause whose satisfaction implies the existence of a new execution. We have designed SATCheck's encoding so that it is simple to generate such a clause. The clause simply needs to evaluate to true when a branch or an uninterpreted function takes on a novel value. These conditions are explicitly represented in our encoding. The interleaving that drives the program to a novel execution can then be recovered from the SAT solution — the truth assignments for the interleaving variables directly encode the desired interleaving.

2.3 Iterating Concrete Executions

Having encoded a SAT formula whose satisfaction implies a novel execution, SATCheck calls the SAT solver to request a satisfying assignment of that formula and converts that assignment back into an execution interleaving. It repeats the execute/encode/solve loop until it exhausts all possible novel behaviors.

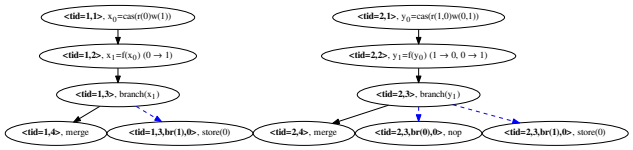

Figure 3. Second execution adds new behaviors for tid=2, including new uninterpreted function outputs and a store node.

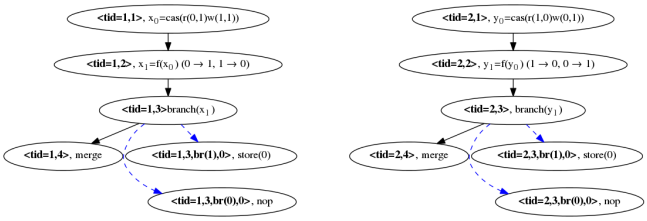

Figure 4. Complete event graph after third execution, summarizing all behaviors for given program input.

In our example, SATCheck might learn from the SAT solver that the branch in Thread 2 would take a different direction if Thread 2 executes `trylock` first. The satisfying assignment encodes properties of the interleavings which would cause this novel behavior. In one such interleaving, (1) Thread 2 executes the CAS operation in `trylock`, (2) Thread 2 executes the `store` in `unlock`, (3) Thread 1 executes the CAS operation in `trylock`, and (4) Thread 1 executes the `store` in `unlock`. SATCheck then executes the program under this new interleaving and incorporates this new execution into its event graph. Figure 3 presents the resulting event graph.

From this graph, SATCheck observes that it still has not explored an execution in which Thread 1 skips the body of the `if` statement and may not have evaluated the uninterpreted functions on all possible input values. Thus, SATCheck repeats the process. The new solution from the SAT solver generates the following interleaving: (1) Thread 2 executes the CAS operation in its `trylock`, (2) Thread 1 executes the failed CAS operation in its `trylock`, and (3) Thread 2 executes the `store` in its `unlock`.

After SATCheck executes that interleaving, it has explored all possible branches in the program. The final event graph shown in Figure 4 reflects the third interleaving and shows that all branches are taken in all directions.

However, it is not immediately obvious from the event graph that SATCheck has evaluated the uninterpreted functions on all possible input values. SATCheck thus generates a new query for the SAT solver. The SAT solver then reports that it is not possible to evaluate the uninterpreted function nodes on any new inputs.

At this point, SATCheck terminates, having explored all possible behaviors for the given program input. Furthermore, SATCheck has produced a SAT model of all possible executions.

$$
\begin{aligned}
\text{cond\_branch}_n(\langle i_0, ..., i_k \rangle) &= \langle i_0, ..., i_k, n, 0 \rangle \\
\text{merge}_l(\langle i_0, ..., i_{l-1}, ..., i_k \rangle) &= \langle i_0, ..., i_{l-1} + 1 \rangle \\
\text{loop\_enter}(\langle i_0, ..., i_k \rangle) &= \langle i_0, ..., i_k, 0 \rangle \\
\text{loop\_exit}_l(\langle i_0, ..., i_{l-1}, ..., i_k \rangle) &= \langle i_0, ..., i_{l-1} + 1 \rangle \\
\text{others}(\langle i_0, ..., i_k \rangle) &= \langle i_0, ..., i_k + 1 \rangle
\end{aligned}
$$

Figure 5. Rules for Updating Operation Tuples

3. Event Graph

As SATCheck executes a program, it dynamically builds an event graph representation for each thread which captures the thread's behavior over observed program executions. The event graph consists of a set of nodes N and set of edges $E \subseteq N \times N$. Nodes in the event graph represent dynamically executed program operations—there is a unique node for each dynamic instance of a program operation in an execution (even if nodes correspond to the same static source code operation). The encoding is SSA-like: all nodes (except phi nodes) accept inputs from exactly one node in the event graph.

The nodes represent the following program operations: (1) stores, (2) loads, (3) atomic RMWs, (4) conditional branches, (5) control flow merges from conditional branches, (6) uninterpreted function invocations, (7) equals comparisons, (8) loop entries, (9) loop exits, and (10) phi functions.

When operations in these nodes take values from variables or temporaries, an event graph node records the source node for that value. SATCheck uses phi function nodes to ensure that there is always exactly one source node for each variable for non-phi nodes.

To merge a new execution into the event graph, SATCheck must match equivalent operations from different executions. SATCheck does this by defining execution point (EP) tuples. A thread starts with the EP $\langle \mathbf{tid}, \mathbf{0} \rangle$. Figure 5 presents the rules for updating a thread's operation tuple during an execution. SATCheck identifies operations at the same EP with the same event graph node, even across executions.

Event graph nodes record all values that SATCheck has observed as output for the operation that corresponds to the node. For function nodes, SATCheck also records, for each previously observed assignment of values to inputs, the output value of the function.

3.1 Potential Value Set Analysis

The SAT translation process begins with a fixed point computation of potential value sets for each variable over the event graph. These sets enable the translation to fix encodings for the generated SAT instance. As discussed in the previous section, potential value sets do not include all possible values for a variable. They only include all possible values under

the current set of executions. SATCheck lazily generates the remaining possible values in future iterations as it encounters new program behaviors that create those values.

Encoding all possible values of program variables would increase the size of the SAT encoding. However, encoding only previously-observed values would affect the SAT solver's ability to reason about new executions (requiring more iterations). Thus, we compute a potential value set for the operation corresponding to each event graph node that includes (1) values we have observed for that operation from previous executions, (2) values that could be propagated by rearranging the reads-from relation between loads and stores without evaluating uninterpreted functions on new values, and (3) values that could propagate through phi functions. Our computed value sets encoded existing behaviors and did not require too many needless iterations.

- **Load/Stores/CAS:** The potential value set for the value read by a load is the union of the potential value sets for the values written by all stores (or RMW operations) that may write to an address that the load may read from. To determine where a load may read from or where a store may write to, we use the potential value sets of the addresses.
- **Atomic Add:** SATCheck approximates the output values written at an atomic add using the values SATCheck has observed from the atomic add in previous executions. This ensures termination, which would not otherwise be guaranteed by using the potential value set for the add's inputs. This design choice has a cost—SATCheck must treat new outputs of atomic adds as new behaviors that the SAT solver explicitly searches for executions to generate.
- **Functions:** The potential value set for the output of a function is the set of values it has generated in previous executions.
- **Equals:** The potential value set for an equals operation is true or false.
- **Phi Functions:** The potential value set for a phi function is the union of the potential value sets of all of its inputs.

3.2 Partitioning Memory Operations

SATCheck partitions load, store, and RMW actions such that if two actions can access the same memory address, they are in the same partition. For this computation, we use the set of addresses computed by the potential value set analysis. All memory operations from the same partition share the same value and address encodings.

4. Encoding the Event Graph into SAT

We next discuss how SATCheck encodes the event graph into SAT. The encoding has two components: (1) SATCheck first chooses SAT variables to encode executions (Section 4.1); and (2) SATCheck formulates SAT clauses to ensure that executions are valid (Section 4.2).

4.1 Representing Executions with SAT Variables

Encoding Interleavings (Execution Order) For any two memory operations $n_i, n_j \in N_{\text{memory}}$, where n_i is performed by thread i and n_j is performed by thread j: if $i < j$, and n_i and n_j are not ordered by thread creation or thread joins, then SATCheck creates a SAT variable v_{n_i,n_j} that is true if n_i is executed before n_j (i.e., $n_i \xrightarrow{sc} n_j$) and false if n_j is executed before n_i. This SAT interleaving variable effectively encodes the execution interleaving. Note that execution graphs describe the behaviors of multiple executions—a given store may not be executed in a given execution. Our encoding orders all memory operations including those that are not executed—memory operations that are not executed simply do not have any effects.

Encoding Control Flow The event graph summarizes all observed executions. For example, if SATCheck has explored both sides of a conditional branch, the event graph will contain the events for both sides. A solution to the SAT formula describes a single execution, and thus the SAT formula must encode all potential executions' control flows.

In a given execution, a conditional branch operation can either: (1) not be executed, (2) take the same direction as we have observed in a previous execution, or (3) take a new direction down the branch (if there exist unexplored directions for that branch).

Conditional branches in SATCheck are used to model `if` statements and `switch` statements, and thus support more than 2 directions. Our encoding uses one state to model the case that the branch was not executed and one state for each of the possible directions the branch has been observed to take. For branches that still have unexplored directions, our encoding also uses one state to model the branch taking a new direction.

Thus, for a conditional branch *br* with m possible directions of which we have observed n directions, we encode $r = 1 + \min(m, n + 1)$ possible behaviors. SATCheck uses $\lceil log_2(r) \rceil$ SAT variables to model each of these possible behaviors.

Values of Program Variables As mentioned in Section 3.1, encoding all possible values for program variables significantly increases the size of the encoding. Instead, SATCheck first computes the potential values S_v for a variable using the potential value analysis from Section 3.1. It then creates $\lceil log_2(|S_v|) \rceil$ SAT variables to encode the value of variable v. The SAT variable encodes the variable v's value as an index into S_v.

For the output of uninterpreted functions or atomic adds, we add a special value to S_v to encode new outputs that have not yet been observed.

Encoding Memory Operations For each memory operation partition p, we have both a set of addresses A_p that the operations may access and a set of values V_p that a load may read or that a store may write. SATCheck uses $\lceil log_2(|A_p|) \rceil$

SAT variables to encode addresses and $\lceil log_2(|V_p|)\rceil$ SAT variables to encode values. SATCheck encodes addresses and values by the binary encoding of their index in the corresponding value set.

Note that memory operations may operate on SAT variables expressed in different encodings. For example, a store operation may take its input value and address from operations that use a different encoding than the store's memory partition. SATCheck generates implication constraints to translate between different encodings. Load operations induce a set of SAT variables for values read and addresses read from, each in an appropriate encoding for the operation. Store operations induce SAT variables for values written and addresses written to.

Encoding the Reads-From Relation For each load ℓ, SATCheck computes stores R_ℓ whose potential address set has a nonempty intersection with the potential address set for load ℓ. SATCheck then uses $\lceil log_2(|R_\ell|)\rceil$ SAT variables to encode which store load ℓ reads-from. SATCheck encodes the reads-from relation as the binary encoding of the store's index in the set R_ℓ.

4.2 SAT Clauses Ensuring Valid Executions

We next describe how SATCheck encodes executions in terms of constraints on the SAT variables described in the previous section.

Branch Constraints As the event graph summarizes the behavior of all executions SATCheck has previously explored, a given execution will typically not execute all the events in the event graph. We next describe the constraints that capture the path of a given execution through the event graph.

SATCheck assumes that conditional branches have a nested structure. Thus at each event e in the event graph, we can compute a nested stack s_e of conditional branches and directions that were taken to reach the given event node. For each conditional branch b, we have an input variable v_b, and potentially a required preceding conditional branch b' that took direction d (e.g., branch b is only reachable when branch b' takes direction d).

We can encode the behavior of branch b using the following two constraints:

1. If branch b' did not take direction d, then branch b was not executed.
2. If branch b' took direction d, then branch b takes the direction v_b specified by its input variable.

Transitive Ordering Constraints for Interleavings We next describe the transitive ordering constraints that ensure that the ordering $\xrightarrow{sc}$, i.e., the execution interleaving, constitute a total order on all memory operations.

For any three memory operations $n_i, n_j, n_k \in N$, SATCheck generates a set of transitive ordering constraints. These constraints capture the following property: $n_i \xrightarrow{sc} n_j \wedge n_j \xrightarrow{sc} n_k \Rightarrow n_i \xrightarrow{sc} n_k$.

Load Read-From Constraints We next describe several consistency constraints between the reads-from relation, the $\xrightarrow{sc}$ total order, the values read by loads and written by stores, and the addresses accessed by loads and stores. These constraints are the final component of encoding the execution interleaving—they ensure that the behaviors of memory operations are consistent with $\xrightarrow{sc}$ (the execution interleaving).

SATCheck begins by computing a set of stores that each load may potentially read from. For each such store, SATCheck instantiates the following constraints:

1. **If load b reads-from store a, then store a must be sc ordered before load b:**
$$a \xrightarrow{rf} b \Rightarrow a \xrightarrow{sc} b$$
2. **A load must read the same value as the store it reads-from:**
$$a \xrightarrow{rf} b \Rightarrow \mathsf{value}(a) = \mathsf{value}(b)$$
3. **If a load reads from a store, then both the load and store must access the same memory address:**
$$a \xrightarrow{rf} b \Rightarrow \mathsf{address}(a) = \mathsf{address}(b)$$
4. **To read from a store, it must have been executed:**
$$a \xrightarrow{rf} b \wedge \mathsf{was\text{-}executed}(b) \Rightarrow \mathsf{was\text{-}executed}(a)$$
5. **There cannot be a conflicting store between a load and the store it reads from:**
$$\begin{aligned} a \xrightarrow{rf} b \Rightarrow (\forall c \in \text{Stores}.\ &\neg\mathsf{was\text{-}executed}(c) \vee \\ &\neg a \xrightarrow{sc} c \vee \neg c \xrightarrow{sc} b \ \vee \\ &\mathsf{address}(c) \neq \mathsf{address}(b)) \end{aligned}$$
6. **Every executed load must read from some store:**
$$\mathsf{was\text{-}executed}(b) \wedge \mathsf{is\text{-}load}(b) \Rightarrow \exists a.\ a \xrightarrow{rf} b$$

Load/Store Value and Address Encoding Each load takes an address as an input and each store takes both an address and a value as inputs. SATCheck uses the results of the potential value set analysis to compute the potential values of the input variable (unless a new value is generated via a new behavior). For each potential value of the input variable, SATCheck generates an implication that, if the input variable has value v, then the relevant address or value variables for the load or store must also have value v. As mentioned above, the encodings for the memory operation and for the partitions need not match.

For example, for a store's input value variable v_{input}, SATCheck generates implications of the form $\mathsf{value}(v_{\text{input}}) = n \Rightarrow \mathsf{value}(\text{store's SAT value variables}) = n$ for all n in v_{input}'s potential value set, to convert the encoding of the input variables into the encodings used by the store's memory partition.

Function Constraints For each uninterpreted function, SATCheck stores the relation between specific input value assignments and the corresponding observed output values. For each known assignment of the input values, SATCheck generates an implication that, if the input variables match the assignment, then the function's output value matches the previous output.

If an execution generates a new assignment to the inputs of an uninterpreted function, then none of the implications apply. The function's output can therefore take on any value. Recall that for the output of each uninterpreted function, SATCheck includes a special value that serves a placeholder for a new output value.

Equals Constraints One assumption of SATCheck is that either a fixed program input will lead to a small set of inputs for uninterpreted functions over the set of concurrent executions, or that the outputs of the uninterpreted function on a maximal range of inputs is interesting. In our experience, this is generally true, but equality comparisons (if modeled as uninterpreted functions) can sometimes be an exception.

Concurrent data structure implementations often use equality comparisons on counters to see whether anything has changed. While such computations can feed many combinations of values into the comparison, the only interesting information is whether the inputs to the comparison are equal.

We therefore provide a built-in comparison operation which enables SATCheck to avoid generating all possible inputs for the comparison operation.

Constraints for Atomic Add and CAS operations To simplify SATCheck's treatment of CAS operations, we model a failed CAS operation as a store of the old value. We then handle CAS operations by combining the techniques we have used for load and store operations. The key difference is that SATCheck generates a constraint that sets the value written by the CAS operation to `newvalue` if the value read matches the `oldvalue`, and otherwise simply stores the same value that the CAS read. We assume a strong CAS operation here; it is straightforward to modify the encoding to support weak CAS operations with spurious failures.

Although it is conceptually straightforward to compute the output of an atomic add operation given its inputs during the potential value computation, doing so can prevent the potential value computation from terminating. SATCheck treats atomic add operations as a combination of a load operation, an uninterpreted function invocation, and a store operation.

Yields Like many model checkers, SATCheck uses yields to avoid the problem of unfair schedules causing the program to loop. Break statements out of conditionally-executed loops complicate our treatment of yields: SATCheck may not be aware of their existence when it discovers a yield. Thus, SATCheck generates a constraint that an execution should not contain a yield unless the thread that calls yield first explores a new branch direction.

4.3 Generating New Behaviors via Goal Expressions

SATCheck iteratively generates complete event graph models by recording past program behaviors in the event graph and generating clauses (goal expressions) that, when true, indicate executions that demonstrate new behaviors. There are two ways that the event graph for a program can be incomplete:

- **Untaken Conditional Branch Directions:** If no previous program execution has taken a given direction of a conditional branch, the event graph will miss events reachable via that branch direction.
- **Unknown Behaviors for Uninterpreted Functions or Atomic Add Operations:** If there is an input assignment to an uninterpreted function or atomic add operation that can be generated by an execution and SATCheck has not explored an execution that generates that input assignment, then SATCheck's model of that uninterpreted function is incomplete.

For conditional branches, SATCheck generates a goal expression that evaluates to true if the branch takes a new direction. For uninterpreted functions, SATCheck generates a goal expression that evaluates to true if the function outputs the unknown-output placeholder.

After exploring the entire event graph, SATCheck generates a SAT formula that is true if at least one goal expression is true.

5. Exploring Concrete Executions

After SATCheck encodes the event graph as a SAT formula, it passes this formula to a SAT solver. If the SAT solver finds that the formula is unsatisfiable, then the event graph is complete: it is impossible to construct an interleaving such that uninterpreted functions see new input value assignments or that branches take new directions. Thus, SATCheck has explored all reachable behaviors of the program.

If the SAT solver finds a solution to the SAT formula, the solution can be converted to an execution exhibiting behavior that is not currently modeled. SATCheck converts the SAT solution into paths through the event graph for each thread. Each path traverses a set of memory operations, and the truth assignments for the interleaving variables specify the execution interleaving that generates the desired new behavior. SATCheck represents the interleaving as a set of *wait pairs*. A wait pair consists of two memory operations: a *stop point* and a *notify point*. A thread's execution stops at a stop point until its partner thread has reached the notify point.

The SATCheck scheduler then performs a concrete execution using the wait pairs. Note that the interleaving generated by the SAT formula is only guaranteed to be realizable until the point at which the execution deviates from previous behavior. After the execution exhibits a new behavior, there is

no guarantee that the execution will continue to follow the path modeled by the SAT solution.

To extend a concrete execution, the SATCheck scheduler executes events from a thread's execution until that thread reaches a memory operation. At a memory operation, the scheduler may choose a new thread. SATCheck's scheduler uses a round robin approach to select a new thread for execution, respecting constraints on thread selection imposed by wait pairs.

When an execution deviates from the interleaving specified by the SAT formula due to a new behavior, it is possible that all threads may get stuck waiting at wait pairs. Recall that wait pairs are based on previous executions and may no longer be valid. Hence, if no threads are runnable because of wait pair constraints, SATCheck can ignore these constraints and arbitrarily pick a thread to run.

Note that SATCheck does not choose which new behavior to explore first—it explores whichever new behavior the SAT solver discovers. When SATCheck observes a new behavior, it integrates that behavior into the event graph, ensuring that the corresponding goal does not get generated in the future.

6. Extensions

We next describe several extensions we have implemented to the core SATCheck algorithm to improve performance and to support TSO.

6.1 Field Support

The base algorithm can handle loads and stores to fields through uninterpreted functions. Failing to differentiate the objects containing the generated fields will cause SATCheck to explore executions that actually access all of the objects that each load or store can access.

Instead, explicitly modeling field accesses can greatly reduce the number of executions that SATCheck needs to explore. We have therefore added explicit support for fields. This embeds the address computation directly into the SAT encoding for the memory access. SATCheck then only needs to generate executions that access the fields of interesting structures.

6.2 Sharing Between Instances of Uninterpreted Functions

In many programs, the same uninterpreted functions are accessed many times during an execution. The base algorithm doesn't share information between different dynamic instances of the same uninterpreted function and thus attempts to generate executions that generate the same inputs for different instances of the same uninterpreted function. We have added support for uninterpreted function identifiers that signal to SATCheck that inputs learned from one instance of an uninterpreted function can be shared with other instances.

6.3 Incremental Solving

Many SAT solvers support incremental solving modes in which variations of an initial SAT problem can be solved more efficiently by leveraging clauses learned from the initial problem.

SATCheck can leverage incremental solving capabilities by reusing the same SAT encoding to generate additional executions that achieve goals that were not covered by the previous solutions.

While this optimization often improves performance, it may sometimes harm performance. In general SATCheck must fail to find a solution to a new encoding before it can terminate as the reused encodings do not incorporate newly discovered behaviors. If SATCheck encounters hard incremental SAT problems to solve, the effort may have been better spent on solving the updated SAT constraints that incorporate the newly learned behaviors.

6.4 TSO Extension

Modern x86 processors implement the Total Store Ordering memory model. In the TSO memory model, stores are placed in a store buffer before main memory is updated. This allows loads to be reordered above previous stores from the same thread.

We have implemented TSO support in SATCheck. The key idea is to separate store operations into two components: a locally visible store action and a globally visible update that moves the store from the local store buffer to update shared memory. Abdulla et al [2] use a similar approach to extend DPOR for TSO.

Without loss of generality, SATCheck executes locally visible store actions immediately after the previous load operation from the same thread. Because they are only locally visible, local stores commute with operations from other threads. SATCheck then encodes a search (to determine when the store should become globally visible) into the SAT formula.

Loads may be reordered in front of the update actions for previous stores from the same thread. Thus we introduce ordering variables between loads and the update actions from previous stores in the same thread. We also change the meaning of the existing execution order variables for threads—these variables now model when the update is flushed to shared memory. This does increase the number of potential executions to explore as a given thread may now have more than one operation it can execute before—either an update from the store buffer or a load operation.

TSO also includes a fence that flushes the store buffer and makes all previous stores globally visible. RMW actions on x86 also have the effect of flushing the store buffer. To account for these, we treat loads slightly differently. If the update action for a store has not been evicted from the store buffer, later loads from the same address from that thread must read from the local store buffer. Fence operations and

RMW operations have the effect of flushing the store buffer if executed. We implement this behavior as an implication—if the fence or RMW action is executed, then we force order variables for the updates of previous stores ahead of the fence or RMW to order them before loads after the fence or RMW.

7. Memory Models

Although SATCheck was implemented to support the SC and TSO memory models, its techniques are applicable to more relaxed memory models. It is straightforward to support processor memory models, as long as the output of uninterpreted functions must depend solely on their inputs. Processors must not reorder a store depending on an uninterpreted function before a load that provides the function's input.

Handling language memory models is a more complex issue, as the mainstream language memory models are known to be incompatible with formal reasoning [3, 8, 29] due to out-of-thin-air (OOTA) behaviors [7]. With the addition of reasonable constraints that forbid OOTA behaviors, such as those suggested by [6, 7], it should be possible to adapt the techniques in SATCheck for use in checking programs against relaxed language-level memory models.

8. Test Schedule Generation

An advantage of combining concrete execution with a SAT solver to guide exploration of executions is that, even if the resulting program is too complex to fully analyze, the SAT solver will generate schedules that can be useful for testing. Approaches that are solely based on SAT will either provide a complete answer or generate a query that is too complex for the SAT solver to handle. In that case, the query itself is not of independent interest. Stateless model checking approaches based on DPOR do yield vast numbers of executions, but many of these executions are redundant.

On the other hand, each execution of SATCheck explores some new aspect of the input program—by modifying the program's scheduling, SATCheck either exercises a new control flow path or produces new and potentially interesting inputs to uninterpreted functions. These executions could generate specifications of interesting test cases in an appropriate test case specification language [12].

9. Instrumentation

While SATCheck builds models of program execution by observing and guiding its dynamic behavior, it also requires program instrumentation to propagate dependency information and to identify uninterpreted functions, branches, and loops. We have implemented a Clang-based frontend which accepts C code and produces instrumented C code suitable for use with SATCheck. Our frontend generates all of the instrumentation for the benchmarks that we present in Section 10.

Although the current implementation is a research prototype, it is based on the industrial-strength Clang frontend. We have started with benchmarks in idiomatic C and gotten them through our frontend without difficulty; getting a new benchmark through the frontend may require a modest amount of straightforward benchmark modification or frontend development (if the benchmark uses parts of C that we do not currently handle).

The instrumentation records provenance information for values that come from the heap or depend on heap accesses. Before each heap access, the instrumentation inserts a call to the SATCheck runtime library with identifiers for the access's input state. The runtime library returns an identifier for the output state. At conditional branches, our instrumenter refactors the condition into a temporary variable if necessary and inserts code to notify the SATCheck runtime library about the condition and the direction that the branch eventually takes.

Most importantly, the instrumentation notifies the runtime library about computations on values that come from shared state. After each computation that depends on shared state, the instrumenter generates an accompanying uninterpreted function notification with the identifiers for the computation's inputs and outputs, thus enabling the creation of function constraints as described in Section 4.2.

Example. We continue with an example demonstrating the operation of our instrumenter. Figure 6 presents the uninstrumented code for the read routine for the seqlock benchmark in Section 10, while Figure 7 presents the output of our instrumenter. The compiler frontend inserts `MCID` variables, which are used by the model checker to represent dependences.

In Figure 6, line 5 contains an `if` statement whose condition needs refactoring. The instrumenter pulls out the condition into variable `_cond30` and reports an uninterpreted function to the model checker at line 9 of Figure 7. The instrumenter also adds branch annotations at lines 11 and 15 and merge annotations at lines 13 and 32. Line 16 (and many others) are shared variable access notifications added by the instrumenter. Finally, the instrumenter inserts a custom uninterpreted function, `MC2_equals`, for the equality condition at line 11 of Figure 7.

Implications. Our front-end enables the model checking of realistic concurrent C code to a scale beyond that achieved by previous techniques. It requires that the code to be verified use a fixed set of primitives to perform shared memory operations (`loads`, `stores` and `rmws`, or read-modify-writes). The user must provide a driver that exercises the functionality of interest in the code to be checked. The driver must also supply all needed inputs.

Our use of uninterpreted functions enables the wrapping of arbitrary binary blobs (including library calls). If some code to be verified calls a binary blob, or library function, that is known to be pure, then the insertion of an uninterpreted function after the binary blob will enable the verification of that code.

```
1  int seqlock_read() {
2      int res;
3      int old_seq = load_32(&_seq); // acquire
4
5      if (old_seq % 2 == 1) {
6          res = -1;
7      } else {
8          res = load_32(&_data);
9          int seq = load_32(&_seq);
10
11         if (seq == old_seq) {
12             ;
13         } else {
14             res = -1;
15         }
16     }
17     return res;
18 }
```

Figure 6. Uninstrumented seqlock read.

10. Evaluation

We have implemented SATCheck and have made it available as open source at the following URL:

`http://plrg.eecs.uci.edu/satcheck/`

A Vagrant environment which reproduces our experimental setup is also available at:

`https://github.com/patricklam/satcheck-vagrant`

We evaluated the performance of SATCheck on a number of benchmarks and compare it to previous work. Our results (see Table 1) show that SATCheck greatly outperforms previous work: it can explore larger problem sizes and runs more quickly than the related work. We ran our evaluations on identically configured Ubuntu Linux 14.04 machines with Intel Xeon E3-1246 v3 CPUs and 32GB of RAM. We ran each tool on each benchmark with a timeout of 1 hour. We wrote the benchmark drivers so that they would take no inputs; the behavior of some benchmarks do depend on scheduling, i.e. on whether a lock is available or not at the time of the request. Most of the benchmark drivers start 2 threads; exceptions are the CAS spinlock, where we started up to 90 threads, and seqlock, where we used 3 threads. Some runs failed in less than an hour (e.g. due to solver problems). This section presents the results from the sequentially consistent (SC) memory model. Appendix A presents the results from the Total Store Ordering (TSO) model.

We compare SATCheck with:

1. CDSChecker's implementation [24] of Flanagan and Godefroid's dynamic partial order reduction algorithm [16] that incorporates support for sleep sets [17] as described in the addendum [15]. We use sequentially consistent atomic operations in CDSChecker, implemented directly using DPOR and sleep sets.
2. Nidhugg's implementation of source DPOR [2]. We present results for both Nidhugg's SC and TSO memory models.

```
1  int seqlock_read(MCID * retval) {
2      MCID _mres; int res;
3      MCID _mold_seq;
4      _mold_seq=MC2_nextOpLoad(MCID_NODEP);
5      int old_seq = load_32(&_seq); // acquire
6
7      MCID _br30;
8      int _cond30 = old_seq % 2 == 1;
9      MCID _cond30_m = MC2_function_id(31, 1, ↩
           sizeof(_cond30), _cond30, _mold_seq);
10     if (_cond30) {
11         _br30 = MC2_branchUsesID(_cond30_m, 1, ↩
               2, true);
12         res = -1;
13         MC2_merge(_br30);
14     } else {
15         _br30 = MC2_branchUsesID(_cond30_m, 0, ↩
               2, true);
16         _mres=MC2_nextOpLoad(MCID_NODEP),
17         res = load_32(&_data);
18         MCID _mseq;
19         _mseq=MC2_nextOpLoad(MCID_NODEP);
20         int seq = load_32(&_seq);
21         MCID _br31;
22         MCID _cond31_m;
23         int _cond31 = MC2_equals(_mseq, (↩
               uint64_t)seq, _mold_seq, (uint64_t)↩
               old_seq, &_cond31_m);
24         if (_cond31) {
25             _br31 = MC2_branchUsesID(_cond31_m,↩
                   1, 2, true);
26             MC2_merge(_br31);
27         } else {
28             _br31 = MC2_branchUsesID(_cond31_m,↩
                   0, 2, true);
29             res = -1;
30             MC2_merge(_br31);
31         }
32         MC2_merge(_br30);
33     }
34     *retval = _mres;
35     return res;
36 }
```

Figure 7. Instrumented seqlock read; instrumentation calls have a MC2 prefix.

3. An enhanced version of CheckFence that has been extended to support atomic addition operations to efficiently support our benchmarks. CheckFence uses an iterative lazy algorithm to determine loop bounds. We configured CheckFence for the SC memory model.

We evaluated SATCheck on these benchmarks:

CAS spinlock This benchmark (seen in Section 2) uses a compare-and-swap instruction to acquire a lock, and a store instruction to release the lock.

To test this benchmark we create two threads that both attempt to acquire and then release the lock. We vary the number of times each thread attempts to acquire and release the lock from 1 time up to 250 times.

Figures 8 and 13 present the results for this benchmark. For SC, SATCheck was able to model check a test in which 2 threads attempted to acquire and release a lock 250 times. We do not report results for CheckFence for more than 60 trylock/unlock pairs as it was unable to analyze the execution

Table 1. SATCheck scales to larger maximum problem sizes than previous tools.

Max problem sizes for:	CDSChecker	CheckFence (SC)	CheckFence (TSO)	Nidhugg (SC)	Nidhugg (TSO)	SATCheck (SC)	SATCheck (TSO)
CAS Spinlock	8	40	40	8	8	250	100
MSQueue	7	8	8	8	8	13	13
Linuxrwlock	7	15	15	6	6	20	20
Dekker	5	10	10	5	4	30	40
Seqlock	3	25	25	3	3	60	60

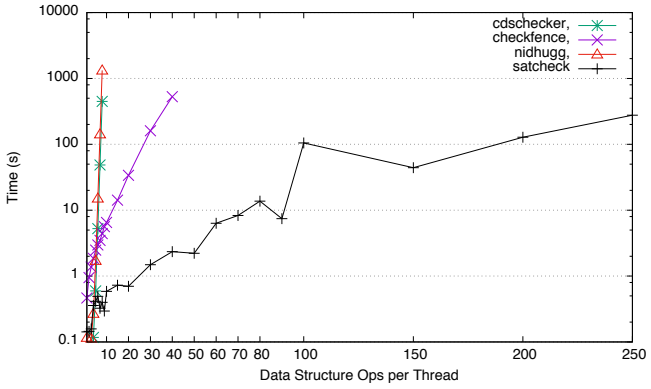

Figure 8. SC CAS spinlock: SATCheck scales to 4× more data structure operations and runs 10×–347× faster than next best tool, CheckFence. (Lower is better, y-axis is log scale.)

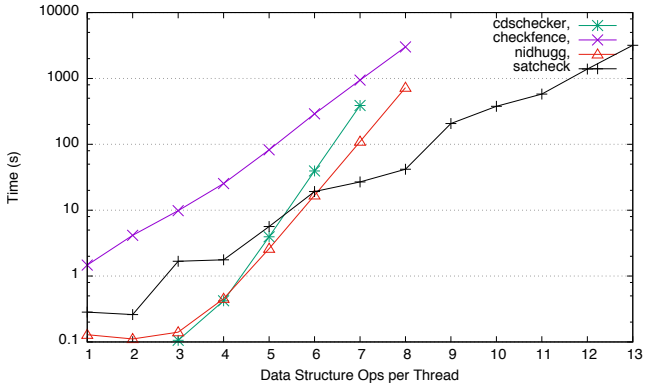

Figure 9. SC MSQueue: SATCheck scales to 62% more data structure operations in an hour and runs up to 17× faster than Nidhugg. (Lower is better, y-axis is log scale.)

for 70 pairs. The DPOR implementations were only able to scale up to 8 trylock/unlock pairs.

We also explored scaling up the number of threads for this benchmark. See Figure 18 in the Appendix for results: SATCheck scales similarly when either data structure ops per thread or number of threads increase. In one hour, SATCheck verified a 90-thread run.

MSQueue We ported the Michael & Scott lock-free queue [22] from CheckFence's version to CDSChecker and SATCheck. The benchmark starts two threads. One thread enqueues values, while the other dequeues values. The standard version makes extensive use of pointer arithmetic—outside of CheckFence's limited support for pointer arithmetic. Although SATCheck supports the standard version, we report results from the ported version, which omits the pointer arithmetic, to enable comparisons with CheckFence.

Figures 9 and 14 present results for MSQueue. SATCheck was able to verify 13 operations per thread within the hour. CDSChecker was able to verify 7 operations per thread and CheckFence and Nidhugg were able to verify 8 operations per thread within the allocated hour.

Linux reader-writer lock A reader-writer lock allows multiple readers or a single writer to hold the lock. No reader can share the lock with a writer. We ported our reader-writer lock benchmark from an implementation in the Linux kernel. However, the kernel implementation is written in assembler for various platforms. We translated the implementation into standard C code.

To test the reader-writer lock, our test driver runs two identical threads, with a single `rwlock_t` protecting a shared variable v. Each thread repeatedly does a trylock on the lock and then frees it.

CheckFence was unable to run the unmodified version of the Linux reader-write locks as the bias value used by the locks exceeded the built-in range threshold for CheckFence. The reported results for CheckFence are for a modified benchmark version that uses a smaller bias.

Figures 10 and 15 present the results for this benchmark. SATCheck can analyze 20 pairs. CDSChecker can analyze 7 trylock/unlock pairs, Nidhugg 6, and CheckFence 15 pairs within the allocated hour.

Dekker Dekker implements a simple critical section using Dekker's algorithm [31], where a pair of non-atomic data accesses are protected from concurrent data access. Our driver for this benchmark is simply two threads each repeatedly calling the critical section routine. Figures 11 and 16 present results for Dekker. SATCheck can verify up to 30 operations, while CheckFence only verifies up to 10 operations; beyond that number, it returns "Inconclusive". CDSChecker and Nidhugg were only able to check up to 5 operations per thread within the allocated hour.

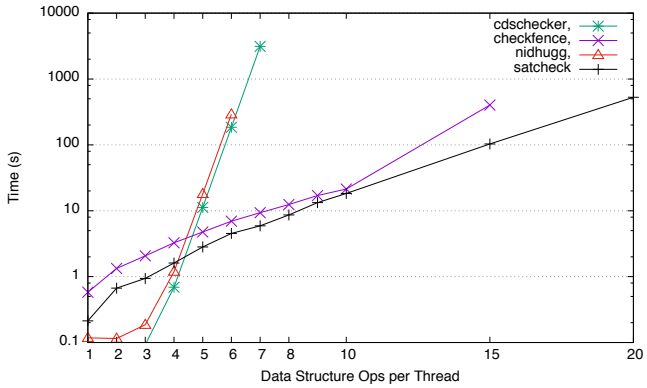

Figure 10. SC Linux RW Lock: SATCheck scales to 33% more data structure operations than CheckFence and takes about the same time, except at 15 operations, where SATCheck is 3.8× faster. (Lower is better, y-axis is log scale.)

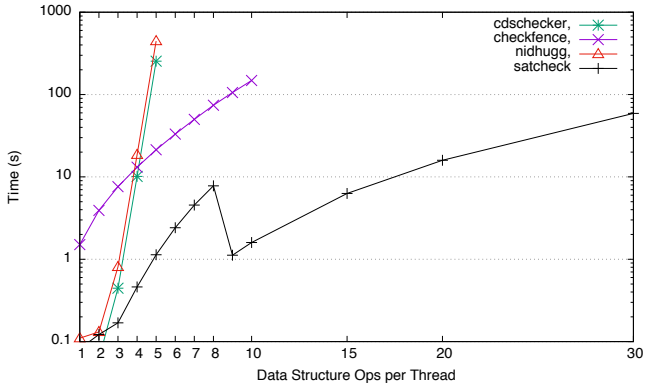

Figure 11. SC Dekker Critical Section: SATCheck scales to 3× more operations than CheckFence and runs 10×–86× faster. (Lower is better, y-axis is log scale.)

Seqlock Seqlocks are used in Linux to avoid writer starvation. They allow writers to update without worrying about readers and thus allow the kernel to communicate with user-space applications.

To test this benchmark, we run one writing thread and two reading threads. Figure 12 and 17 present the results. The seqlock benchmark scales to fewer data structure operations per thread because there are more threads. SATCheck could verify up to 60 operations per thread. CDSChecker and Nidhugg were only able to verify up to 3 operations per thread, while CheckFence could verify up to 40 operations per thread. At 40 operations per thread, SATCheck was 2.6× faster than CheckFence.

Discussion. Our results show that SATCheck runs much faster and hence scales better than previous tools. The main reason for its performance is that it uses a concolic execution approach to finding novel behaviors. It therefore does not need to explore redundant executions, nor does it need to encode values that do not occur in observed executions.

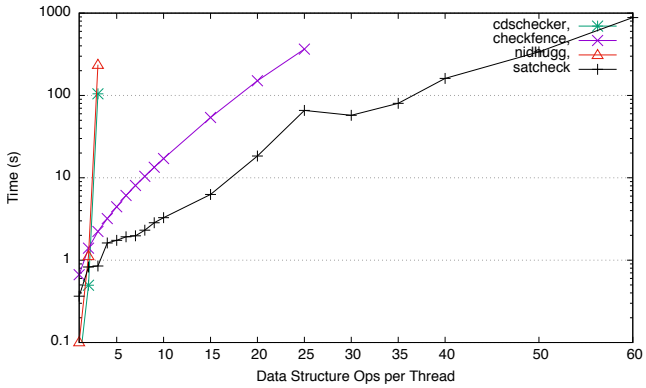

Figure 12. SC Linux Seqlock: SATCheck scales to 1.5× more data structure operations than CheckFence and runs 2×–9× faster. (Lower is better, y-axis is log scale.)

Because most of SATCheck's execution time is in the SAT solver, reported running times aren't perfectly regular. Generally, its performance scales as expected when we increase the number of data structure operations.

11. Related Work

In the Introduction, we described three approaches to model checking concurrent data structures: explicit state, stateless, and SAT-based. Our approach is SAT-based but uses concrete program executions to guide a search for novel behaviors. State-based model checkers such as SPIN [19] can debug concurrent data structure implementations. Inspect combines stateless and stateful model checking to model-check C and C++ code [30, 33, 34]. Related approaches include CHESS [23], which finds and reproduces concurrency bugs in C, C++, and C# by systematically exploring thread interleavings. However, it can miss concurrency bugs as it does not explore all thread interleavings.

CheckFence [9] is the most closely related approach to ours. It focuses on verifying concurrent data structure implementations for relaxed memory models, including usage of memory fences. The primary difference between our approach and CheckFence is that our approach uses the SAT solver to guide concrete program executions, while CheckFence encodes entire abstract program executions with SAT. Due to its static approach, CheckFence must lazily unroll loops when the SAT solver indicates that it is possible for a loop to execute more times than the current unrolling. Like us, CheckFence also uses a range analysis to compute ranges for variables. However, because CheckFence's range analysis operates on the static program, we found that it often computes too large of a range for a variable. CheckFence then refuses to analyze the program. Also, as CheckFence must encode the entire program into SAT, it requires source for the entire program, and the program must be amenable to compilation to SAT.

MemSat [28] uses constraint solvers to reason about programs under weak memory models. It targets very complex memory models (Java Memory Model) and very simple programs. Our work explores a new approach for handling large test cases.

DPOR [16] and ODPOR [1] are two dynamic approaches to partial order reduction that reduce the number of program executions to be explored by a stateless model checker. These approaches avoid exploring executions that can be generated by reordering commuting operations in some other explored execution. Recent work has extended the DPOR algorithm to handle the TSO and PSO memory models [2, 35]. Our work can be viewed as exposing the interleaving operations to the SAT solver, iteratively building a model of the executions, and letting the SAT solver's heuristics avoid the redundant behaviors (rather than doing the partial order reduction via reordering ourselves). Alternatively, our work can also be viewed as dynamically generating a SAT formula that describes all possible executions.

Researchers have recently proposed maximal causality reduction to improve on POR [20]. The key insight behind maximal causality reduction is that a thread's behavior does not depend on the specific stores that the thread's loads take its values from, but rather the values that these loads read. This approach uses a constraint solver to generate executions in which the loads of a thread read different combinations of values than previously explored executions. It conservatively assumes that any store may depend on previous loads, and thus must explore vastly more executions that SATCheck. The author has not made an implementation available, but a quick calculation reveals that it will be many orders of magnitude slower than SATCheck. For example, for the 250 pair spinlock example that SATCheck analyzes in 4.5 minutes, the maximal causality reduction approach would need to explore at least 2^{250} executions.

Researchers have developed a tool based on Nitpick for translating C/C++11 code to SAT to model check litmus tests [5]. Litmus tests are small tests that contain only a handful of memory operations. Their work focuses on automatically building SAT formulas directly from a formalization of the memory model; it does not focus on tool performance.

Industry tools like IBM ConTest tool support testing concurrent software. While such tools may increase the likelihood of finding races, they are not exhaustive. Like the related work in our field, SATCheck definitely explores all behaviors for a given input.

Several tools detect data races in code that uses standard lock-based concurrency control [11, 13, 14, 21, 26]. These tools generally take one of two approaches: (1) they verify that all accesses to shared data are protected by a locking discipline or (2) they verify that a happens-before relation separates conflicting accesses. Another way to mitigate potential data races is the approach proposed by stable and deterministic multithreading systems [4, 10, 25, 32], which constrain the allowed interleavings. Work on data race detection and stable multithreading systems is largely orthogonal to SATCheck, since SATCheck seeks to verify data structures that leverage low-level atomics to access memory without the use of locks.

12. Conclusion

Threads commonly communicate with each other through concurrent data structures. Developing correct concurrent data structure implementations is known to be challenging and testing tools are critical for finding implementation bugs.

SATCheck leverages concrete executions to build an event graph model of concurrent code and uses a SAT solver to guide executions towards discovering novel behaviors. SATCheck scales better than other tools that leverage concrete execution while avoiding the need to compile the entire program to SAT.

Acknowledgments

This project was partly supported by the National Science Foundation under grants CCF-0846195, CCF-1217854, CNS-1228995, and CCF-1319786. We would also like to thank the anonymous reviewers for their helpful comments.

References

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

[2] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. Stateless model checking for TSO and PSO. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2015.

[3] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013.

[4] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for c/c++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, 2009.

[5] J. C. Blanchette, T. Weber, M. Batty, S. Owens, and S. Sarkar. Nitpicking C++ concurrency. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, 2011.

[6] H.-J. Boehm. N3786: Prohibiting "out of thin air" results in C++14. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3786.htm`, September 2013.

[7] H.-J. Boehm and B. Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, 2014.

[8] H.-J. Boehm et al. N3710: Specifying the absence of "out of thin air" results (LWG2265). `http://www.open-std.org/`

jtc1/sc22/wg21/docs/papers/2013/n3710.html, August 2013.

[9] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: Checking consistency of concurrent data types on relaxed memory models. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[10] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[11] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[12] T. Elmas, J. Burnim, G. Necula, and K. Sen. CONCURRIT: A domain specific language for reproducing concurrency bugs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[13] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.

[14] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[15] C. Flanagan and P. Godefroid. Addendum to dynamic partial-order reduction for model checking software. `http://users.soe.ucsc.edu/~cormac/papers/popl05-addendum.pdf`.

[16] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the Symposium on Principles of Programming Languages*, January 2005.

[17] P. Godefroid. Partial-order methods for the verification of concurrent systems: An approach to the state-explosion problem. *Lecture Notes in Computer Science*, 1996.

[18] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the Symposium on Principles of Programming Languages*, 1997.

[19] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2003.

[20] J. Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[21] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H. Boehm. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.

[22] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, 1996.

[23] M. Musuvathi, S. Qadeer, P. A. Nainar, T. Ball, G. Basler, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, 2008.

[24] B. Norris and B. Demsky. CDSChecker: Checking concurrent data structures written with C/C++ atomics. In *Proceeding of the 28th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2013.

[25] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[26] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computing Systems*, 15:391–411, Nov. 1997.

[27] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[28] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: Checking axiomatic specifications of memory models. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[29] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, 2013.

[30] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. *ATVA LNCS*, (126–140), 2008.

[31] A. Williams. Dekker's algorithm implementation. `http://www.justsoftwaresolutions.co.uk/threading/`. Dec. 2012.

[32] J. Yang, H. Cui, J. Wu, Y. Tang, and G. Hu. Making parallel programs reliable with stable multithreading. *Communications of the ACM*, 57(3), 2014.

[33] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. In *Proceedings of the 15th International SPIN Workshop on Model Checking Software*, 2008.

[34] Y. Yang, X. Chen, G. Gopalakrishnan, and C. Wang. Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 279–295, 2009.

[35] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

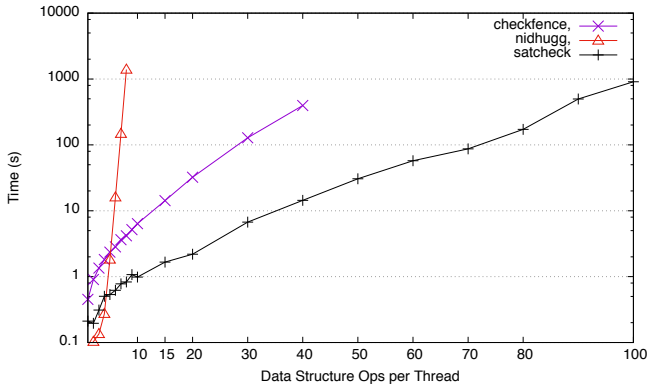

Figure 13. TSO CAS Spinlock Results (lower is better, y-axis is a log scale)

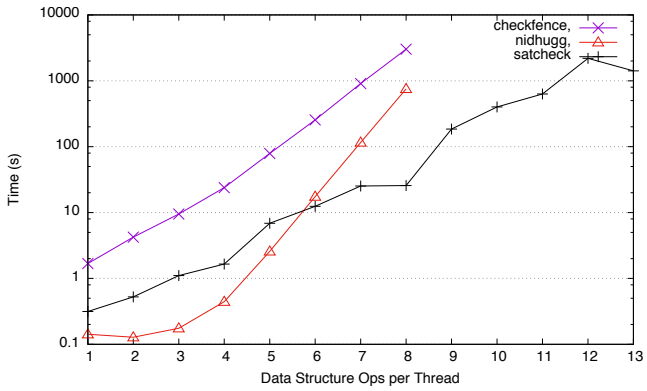

Figure 14. TSO MSQueue Results (lower is better, y-axis is a log scale)

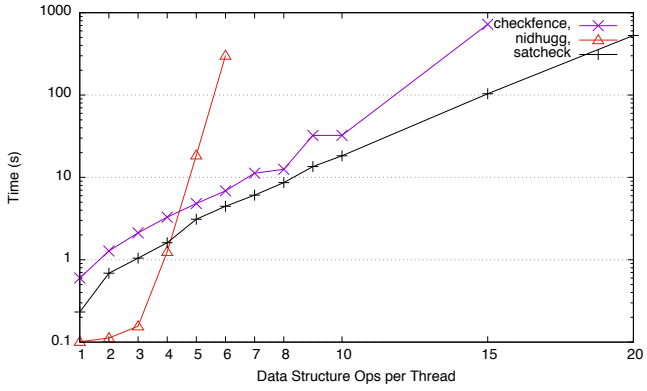

Figure 15. TSO Linux RW Lock Results (lower is better, y-axis is a log scale)

A. TSO Results

In Section 10 we presented results under the sequentially consistent (SC) memory model. We have also collected results for Total Store Ordering (TSO) and present them in Figures 13 through 17 of this Appendix.

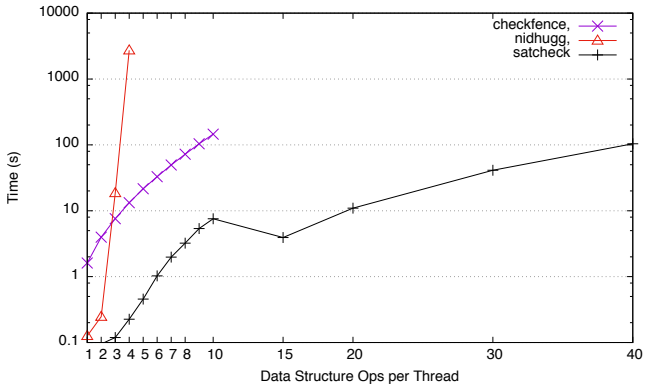

Figure 16. TSO Dekker Critical Section Results (lower is better, y-axis is a log scale)

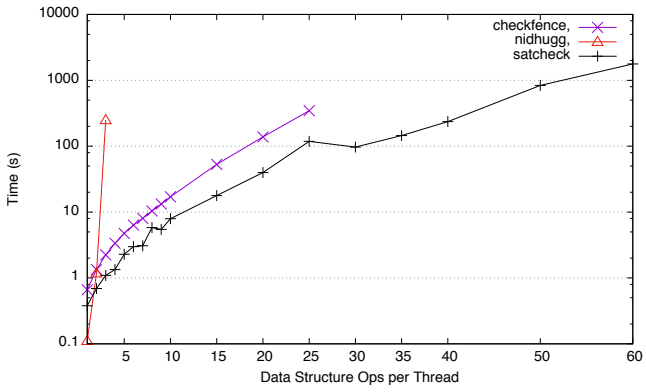

Figure 17. TSO Linux Seqlock Results (lower is better, y-axis is a log scale)

B. Scaling Threads

In Section 10 we presented results that keep the number of threads fixed and scale the number of operations per thread. We also explored scaling the number of threads for the CAS Spinlock: each thread performs one pair of lock and unlock operations.

Figure 18 presents the result of this experiment. The results show that SATCheck scales well with the addition of extra threads.

C. Soundness

Theorem C.1 (Statement Reachability)**.** *If there exists some execution e that can reach a statement st, then SATCheck will explore some execution that executes statement st.*

Proof Sketch. By contradiction. Suppose that SATCheck does not explore any execution that executes statement st. Consider the execution e. There must exist some conditional branch in e that SATCheck has not explored, or it would have reached the statement st.

Consider the first such unexplored conditional branch b or unexplored input i to an uninterpreted function in execution

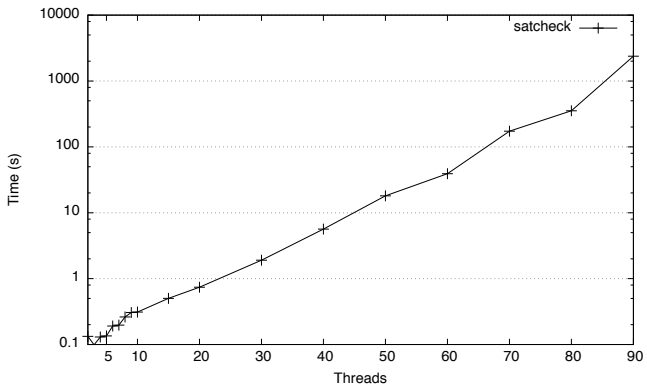

Figure 18. SATCheck analyzes a 90-thread run of Linux Locks (SC) in one hour. (lower is better, y-axis is a log scale)

e. By the design of SATCheck, the branch b or input i is a goal to SATCheck's SAT formula. Consider an execution prefix e' of the execution e up to the branch br or input i. This execution prefix satisfies the branch br or input i goal of the generated SAT formula — up until the new event, all of e''s behavior is modeled by the event graph. The clauses that model the event graph after the prefix e' are all structured as implications from past events to future events. If none of the conditions in the implications on a past event are satisfied, the constraint is trivially true.

Thus there must be solution to the SAT constraints that drives SATCheck to produce the execution prefix e' and thus SATCheck explores e' unless it first explores some other execution that reaches b or i.

Theorem C.2 (Uninterpreted Function Reachability)**.** *If there exists some execution e that generates input i to uninterpreted function f, then SATCheck will explore some execution that generates input i to uninterpreted function f.*

Proof Sketch. Same proof as Theorem C.1.