Synthesis of Synchronization using Uninterpreted Functions

Roderick Bloem, Georg Hofferek, Bettina Könighofer,
Robert Könighofer, Simon Außerlechner, and Raphael Spörk
Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Austria.

***Abstract*—Correctness of a program with respect to concurrency is often hard to achieve, but easy to specify: the concurrent program should produce the same results as a sequential reference version. We show how to automatically insert small atomic sections into a program to ensure correctness with respect to this implicit specification. Using techniques from bounded software model checking, we transform the program into an SMT formula that becomes unsatisfiable when we add correct atomic sections. By using uninterpreted functions to abstract data-related computational details, we make our approach applicable to programs with very complex computations, e.g., cryptographic algorithms. Our method starts with an empty set of atomic sections, and, based on counterexamples obtained from the SMT solver, refines the program by adding new atomic sections until correctness is achieved. We compare two different such refinement methods and provide experimental results, including Linux kernel modules where we successfully fix race conditions.**

I. Introduction

Concurrency-related bugs form a serious problem in software development. First, concurrent programs are hard to get right due to the large number of possible interleavings of threads. Second, concurrency issues are difficult to detect and to reproduce: faults may only appear in rare cases that are never hit by tests but only in operation. Third, even if detected and reproducible, concurrency errors are difficult to fix. There is the danger of fixing only some but not all symptoms, or even introducing new errors. At the same time, the desired behavior of a concurrent program is typically easy to specify: it should behave as if executed sequentially. This important property is called *serializability*, meaning that any concurrent execution must behave as if all threads were executed one after the other (in some order). In this paper, we present methods to synthesize efficient synchronization in form of atomic sections to ensure serializability. Assertions can be used as an additional (or alternative) specification. Thus, on a high abstraction level, we address the same problem as [7, 24].

Adequate *abstraction* is a key factor in making synthesis of synchronization tractable. Our intuition is that synchronization usually should not depend on the semantics of data operations. Thus, we propose to use abstract data operations by means of *uninterpreted functions*. This is done by replacing all arithmetic operations as well as calls to functions without side-effects by uninterpreted functions during program analysis. This speeds up the synthesis process significantly. However, abstraction may induce spurious counterexamples, which may lead to more and larger atomic sections than actually necessary. One way to address this issue is to allow the user to refine (some) uninterpreted function symbols with fundamental properties like commutativity and associativity. Such properties are important in the context of concurrent programs because different interleavings often apply the same operations in different order (e.g., $(3+4)+5$ vs. $4+(5+3)$).

Building on abstraction by means of uninterpreted functions, we present and compare two synthesis methods. They repeatedly check for counterexamples (executions violating the specification) and add atomic sections until no more counterexamples exist. Counterexamples are computed by a Satisfiability Modulo Theories (SMT) solver, using a Bounded Model Checking (BMC) approach [21]. We unroll loops in the program and guarantee correctness only up to the unrolling depth. First, we present a novel method that we named FixSwitches. It analyzes counterexamples with a heuristic to guess the context switch that causes the problem, and forbids this switch with an atomic section. It does not guarantee minimality of the atomic sections, nevertheless it always produced a minimal solution in all our experiments. The second method, named AtomConstr, is based on [24] and collects constraints for the atomic sections based on the counterexamples: at least one context switch of every counterexample must be forbidden. These constraints are then solved to obtain a global minimum of atomic sections. We implemented our synchronization synthesis approach in a prototype tool called Atoss and present first experimental results. We also compared our methods with several set minimization algorithms (e.g. the QuickXplain algorithm [17]), trying to find a (locally) minimal set of atomic sections that is sufficient to make the program correct. It turns out that FixSwitches and the AtomConstr algorithm scale best, so we do not present these experiments in detail.

Related Work. A lot of work has been done to *verify* concurrent programs [14, 10, 8]. Verification is an important building block in our synthesis method: we use a BMC approach [21] to search for counterexamples. Automatic synthesis of synchronization was first considered in 1981 by Clarke and Emerson [7]. In the last few years, this topic was taken up again, e.g. in [23], [24], [5], [18], and [6]. Vechev et al. [24] abstract the program state using a finite domain and compute counterexamples by explicitly searching

This research was supported by the Austrian Science Fund (FWF) through projects RiSE (S11406-N23) and QUAINT (I774-N23).

through the abstract transition system graph. Then, heuristics decide whether to refine the abstraction or insert an atomic section. The user has to provide a characterization of the good states as specification. In contrast, our approach can take the sequential behavior as implicit specification, it searches for counterexamples symbolically, using an SMT solver, and uses uninterpreted functions for abstraction. Counterexample-guided synthesis is also considered in [5]. Counterexamples are generalized to so-called partial-order traces that represent all counterexamples that lead to the same error. Partial-order traces are eliminated by lock insertion, but also by other semantics-preserving program transformations like instruction reordering. In contrast, we consider counterexamples with an increasing number of context switches, thus we can skip the generalization step. Kahlon [18] considers the problem of fixing concurrency errors once they are detected. Given a set of mutually atomic segments, the algorithm inserts locks around the segments to fix the atomicity violation without introducing new deadlocks. In contrast, our approach does not assume that mutually atomic sections are already given.

Uninterpreted Functions are often used as an adequate mean of abstraction in verification, e.g., in translation validation [20], where a compiler is verified by checking its input and output program for sequential correctness. Another example is proving equivalence between a pipelined and a non-pipelined version of the same processor [4, 3], where the complex datapath elements such as the ALU are abstracted. Abstraction by uninterpreted functions has also been used for synthesizing controllers that avoid concurrency-related problems in pipelined processors [15, 16]. The main difference is that [15, 16] synthesizes controllers whose actions may depend on the current inputs of the system. This amounts to solving formulas of the form $\forall \mathsf{inputs}.\exists \mathsf{control}.\forall \mathsf{outcomes}.\phi$, where ϕ is a correctness criterion. In this paper, we effectively solve problems of the structure $\exists \mathsf{control}.\forall \mathsf{inputs}.\phi$, because in software it is customary to have static synchronization mechanisms that do not depend on the current inputs of a program. This quantifier structure also makes the problem easier and allows us to deal with larger numbers of existentially quantified variables, whereas the approach of [15, 16] scales exponentially w.r.t. this number.

Contributions. In summary, the main contributions of this work are as follows.

- We relieve the user from writing a specification by taking the sequential behavior of the concurrent program as implicit specification.
- To the best of our knowledge, we are the first to use uninterpreted functions as abstraction for synthesis of synchronization. We show that this allows us to handle programs that cannot be handled with finite-domain abstractions.
- We present and compare two methods to infer atomic sections from counterexamples. One is novel and specifically tailored towards our synthesis algorithm, the other one is based on ideas from [24].

Outline. The rest of this paper is structured as follows. Section II discusses preliminaries and establishes notation. Section III presents an illustrating example. Section IV presents the synchronization synthesis algorithms and introduces our abstraction method based on uninterpreted functions. Experimental results are shown in Section V. Section VI concludes and discusses directions for future work.

II. Preliminaries

Concurrent Programs. A concurrent program P is a set of threads $T = \{t_1, \dots t_n\}$. Each thread t_i is represented as a control flow graph $t_i = (b_i, e_i, V_i, E_i)$, where $V_i = \{s_{i1}, \dots, s_{im}\}$ is the set of nodes, $b_i \in V_i$ is a unique start node, $e_i \in V_i$ is a unique end node, and $E_i \subseteq V_i \times L_i \times V_i$ is a set of directed and labeled edges between the nodes. The set of labels L_i is comprised of Boolean expressions ($\mathbb{B}$-expr), defined below. If the control flow graph is cyclic, which means that the program contains loops, we unroll them up to a certain depth to make it acyclic. Each node s_{ij} is labeled by a program statement. For simplicity, we assume that each statement of the concurrent program corresponds to a different node in the graph. Thus, different nodes can be labeled with the same instruction. Edge labels express conditions. An edge $(s, l, s') \in E_i$ means that s' is the successor statement of s if condition l holds. The node e_i does not have a successor. We denote with $\mathcal{G}$ the set of global variables shared between all threads. Furthermore, each thread t_i has a set of local variables $\mathcal{L}_i$. To simplify the presentation, we assume that all program variables range over the same domain $\mathbb{D}$.

We will model concurrent programs as formulas in the quantifier-free fragment of the *Theory of Uninterpreted Functions and Equality* $\mathcal{T}_U$ (`QF_UF`). To do so, we make the following more formal definition of statements and conditions. Let $\mathcal{F}$ be a set of (uninterpreted) functions $f : \mathbb{D}^+ \mapsto \mathbb{D}$, let $\mathcal{P}$ be a set of (uninterpreted) predicates $p : \mathbb{D}^+ \mapsto \mathbb{B}$ with $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$, let $v \in \mathcal{L}_i \cup \mathcal{G}$ be a variable, $f \in \mathcal{F}$ be an uninterpreted function, let $p \in \mathcal{P}$ be an uninterpreted predicate, and let $=$ be the (interpreted) equality predicate. The set of $\mathbb{D}$-expressions and $\mathbb{B}$-expressions is defined as follows.

$$\begin{aligned} \mathbb{D}\text{-expr} \quad &::= \quad v \mid f(\mathbb{D}\text{-expr}^+) \\ \mathbb{B}\text{-expr} \quad &::= \quad p(\mathbb{D}\text{-expr}^+) \mid \mathbb{D}\text{-expr} = \mathbb{D}\text{-expr} \mid \\ & \qquad \neg\mathbb{B}\text{-expr} \mid \mathbb{B}\text{-expr} \vee \mathbb{B}\text{-expr} \end{aligned}$$

A statement is of the form $v := r$, where $v \in \mathcal{L}_i \cup \mathcal{G}$ and $r \in \mathbb{D}$-expr. That is, all statements are assignments; we assume that all function calls have been inlined and do not allow recursion. An edge label $l \in L_i$ is a $\mathbb{B}$-expression. The semantics of statements and conditions on edges are as expected. The labeled edges are such that all statement nodes $s \in V_i \setminus \{e_i\}$ have exactly one successor for every variable valuation (i.e., for a given scheduling, the program is deterministic). We will write $V = \bigcup_i V_i$ for the set of all graph nodes, $V' = \bigcup_i (V_i \setminus \{e_i\})$ for all but the end nodes, and $\mathsf{thread}(s_{ij})$ for the thread t_i to which the statement s_{ij} belongs.

Listing 1 RSA decryption using the Chinese Remainder Theorem (CRT)

```
Input: large primes p, q; ciphertext c; private exponent d;
Output: plaintext in $m_p$
 1: bool $fin_1$=false, $fin_2$=false;
 2: int merged=0, $m_p$=0, $m_q$=0;
 3: procedure THREAD$_1$
 4:   $m_p$=$c^d$ mod p;
 5:   $fin_1$ = true;
 6:   if merged=0 && $fin_2$
 7:     merged=1;
 8:   if merged=1
 9:     $m_p$=crt($m_p$,$m_q$);
10: procedure THREAD$_2$
11:   $m_q$=$c^d$ mod q;
12:   $fin_2$ = true;
13:   if merged=0 && $fin_1$
14:     merged=2;
15:   if merged=2
16:     $m_p$=crt($m_p$,$m_q$);
```

Concurrent Executions and Correctness. An execution of program P is a sequence of statements $\overline{s} = s_1, s_2, \ldots \in V^*$ respecting the program semantics. An atomic section set is a set $A \subseteq V'$. A program execution $\overline{s} = s_1, s_2, \ldots$ respects the atomic section set A if $s_i \in A$ implies $\mathsf{thread}(s_i) = \mathsf{thread}(s_{i+1})$ for all i. That is, if statement s_i is protected by an atomic section, then no thread switch is allowed immediately after the statement. An execution is *sequential* if it respects the atomic section set $A = V'$. In order to define a notion of correctness for concurrent programs, we introduce a function $\mathsf{eval} : V^* \to \mathbb{D}^{|\mathcal{G}|}$, which — given an execution $\overline{s} = s_1, s_2, \ldots$ of P — returns the values of the global variables after the execution. We say that an execution $\overline{s}$ is *correct* if there exists a sequential execution $\overline{s}'$ such that $\mathsf{eval}(\overline{s}) = \mathsf{eval}(\overline{s}')$. A *counterexample* is an incorrect execution. We define a procedure $\mathsf{ce}(A)$ which returns a counterexample that respects an atomic section set $A \subseteq V'$, or the constant None if no such counterexample exists. An atomic section set A is *sufficient* if $\mathsf{ce}(A) = \mathsf{None}$. An atomic section set A is a *local minimum* if it is sufficient and all $A' \subset A$ are not sufficient. An atomic section set A is a *global minimum* if it is sufficient and all A' with $|A'| < |A|$ are not sufficient. Given an execution $\overline{s} = s_1, s_2, \ldots$ of P, we say that a thread switch after statement s_i (with $\mathsf{thread}(s_i) \neq \mathsf{thread}(s_{i+1})$) is *mandatory* if $s_i \notin V'$, i.e., s_i is an end node of some control flow graph. Otherwise, the thread-switch is *non-mandatory*.

III. Illustrating Example

We give an example to demonstrate our approach, in particular the benefits of abstraction with uninterpreted functions. Consider the problem of decrypting an RSA-encrypted message (cf. Listing 1). For efficiency, many cryptographic libraries employ the Chinese Remainder Theorem (CRT) during RSA decryption [19]. As usual, p and q are two large prime numbers, c represents the ciphertext and d is the private decryption exponent. In standard RSA, the message m is obtained by computing $m = c^d \mod p \cdot q$. To speed up the decryption process, Thread 1 computes $m_p = c^d \mod p$ and Thread 2 computes $m_q = c^d \mod q$. After m_p and m_q are found, one of these threads uses the function `crt` to compute the final message (modulo $p \cdot q$) and stores it in m_p. The concurrent program is correct if the final message m_p equals the message obtained by a sequential run of the two threads (in either order).

Without any atomic sections, the following problem could occur. If Thread 1 is interrupted between lines 6 and 7, and Thread 2 executes lines 13–16 in the meantime, Thread 1 will subsequently set `merged` to 1, and execute line 9. However, the merge has already been performed by Thread 2, and doing it a second time results in erroneous output. The problem could be prevented by making lines 6–7 and lines 13–14 atomic sections.

The RSA algorithm uses complex arithmetic functions (modular reductions, exponentiations, etc.) on very large numbers. Modeling this program with linear integer arithmetic is not possible, due to the complex operations involved. On top of that, modeling it with bitvectors is also not feasible, due to the large bitwidths involved. However, when using an abstraction with uninterpreted functions, the resulting SMT formula is rather simple. The line $m_p = c^d \mod p$, for example, reduces to one simple equality between a domain variable and an uninterpreted function instance: $m_p = f_{modexp}(c, d, p)$. Using abstraction with uninterpreted functions, our tool was able to find the minimal set of atomic section in a few seconds (atomic sections spanning lines 6–7 and lines 13–14). Without any abstraction, it would not be possible to verify this program.

Note that the finite-domain abstraction approach presented in [24] cannot deal with this example. One problem is that finite-domain abstractions are not *equality preserving*. They only track properties like the parity of variable values, or whether certain values are in a particular interval. This is usually too coarse to prove the equality of values (without refining the abstraction until all bits of the relevant variables are tracked). Note that this problem also occurs for simple functions such as addition or multiplication.

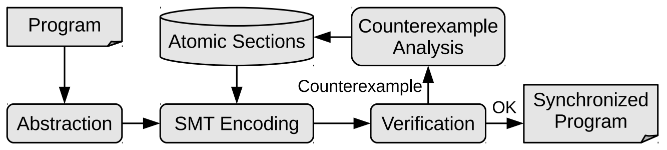

Fig. 1. Overview of our synthesis approach.

IV. Synthesis Approach

The working principle of our synthesis approach is outlined in Figure 1. The main input is a concurrent program P without any synchronization. First, the program is abstracted using uninterpreted functions. This step is explained in Section IV-A. Next a counterexample-guided synchronization refinement loop is entered. There is a database of (candidates for) atomic sections, which is initially empty. Considering these already known atomic sections, we next encode the concurrent verification problem into an SMT formula. Satisfying assignments of this formula correspond to counterexamples, i.e., executions of the concurrent program which violate the specification. The

SMT encoding is discussed in Section IV-B. In the verification step, an SMT solver searches for a counterexample in form of a satisfying assignment of the constructed formula. If a counterexample is found, it is analyzed in order to infer new atomic sections that prevent (at least) this particular counterexample, and we loop back to checking whether the program is correct now. Two different methods for analyzing counterexamples and refining the atomic sections will be presented in Section IV-C and Section IV-D, respectively. If no more counterexamples exist, we have found a set of atomic sections that are sufficient to prevent erroneous executions and the algorithm terminates.

A. Abstraction using Uninterpreted Functions

A program statement updates a variable with a new value that is the result of some computation. The computation can be as simple as an increment, or an inlined addition, but it can also be a call to a complex n-ary function. We observe that in many cases, correctness of a program does not depend on the actual semantics of the functions involved in the computations. For example, if you replace all additions in a correct concurrent program by multiplications, the resulting program still should not depend on the scheduling. The only thing that is relevant to correctness is *functional consistency*, i.e., given the same inputs, a particular statement should always produce the same result.

It might be obvious to use logics based on the theories of linear integer arithmetic, linear real arithmetic, or bitvector arithmetic, which include interpreted and axiomatized symbols encoding addition, multiplication, etc. In fact, loop-free programs can be modeled perfectly using bitvector logic [9]. However, by doing so we burden the SMT solver unnecessarily, because it now has to look for solutions that satisfy all the axioms of the interpreted symbols. In addition to that, more complex operations might not easily (or even not at all) be expressible in terms of the available interpreted functions.

Thus, we suggest to "forget" all the semantics of a statement, and abstract it using uninterpreted functions only. E.g. a statement `a = b + c` becomes $a = f_{plus}(b, c)$, where $f_{plus} \in \mathcal{F}$ is an uninterpreted symbol. In the example presented in Section III, there are two uninterpreted functions that we would need to introduce: $f_{modexp}(\cdot, \cdot, \cdot)$ and $f_{crt}(\cdot, \cdot)$.

However, even though the functions we use are uninterpreted, there are two important properties that are of particular interest in the setting of concurrency: *commutativity* and *associativity*. The reason for that is that different interleavings of threads will lead to a different order of operations. However, knowing that some functions are commutative and associative, it is still possible to prove that the final outcome is the same. One possible way to achieve this is to add those concrete instances of the commutativity and associativity axioms that are actually relevant to a particular example: i.e., state for every pair of variables a, b that $f_{plus}(a, b) = f_{plus}(b, a)$, and similar for associativity. A potentially more efficient way is to add support for commutativity and associativity directly in the congruence closure module of the underlying SMT solver.

Listing 2 C Code

```
 1: int g;
 2: procedure THREAD1            7: procedure THREAD2
 3:   int x = g;                 8:   int y = g;
 4:   x = x + 1;                 9:   y = y + 2;
 5:   g = x;                    10:   g = y;
 6:   x = x + 1;                11:   y = y + 2;
```

Listing 3 SSA Constraints

2:	**procedure** THREAD1	7:	**procedure** THREAD2
3:	$t_1x_1 = t_1g_1$	8:	$t_2y_1 = t_2g_1$
4:	$t_1x_2 = t_1x_1 + 1$	9:	$t_2y_2 = t_2y_1 + 2$
5:	$t_1g_2 = t_1x_2$	10:	$t_2g_2 = t_2y_2$
6:	$t_1x_3 = t_1x_2 + 1$	11:	$t_2y_3 = t_2y_2 + 2$

The theory of how to do this has been outlined in [1]; we are currently working on adding this feature to the Z3 SMT solver [12].

B. SMT Encoding of the Concurrent Verification Problem

This section explains how we encode the concurrent verification problem into an SMT formula such that satisfying assignments correspond to counterexamples. SMT encoding of programs has been addressed before, e.g. in [14] and [11]. We use an encoding called TCBMC [21], with small modifications. The main idea is to limit the maximum number of thread switches while allowing them to be anywhere in the code. This has the advantage that we are able to analyze counterexamples with an increasing number of thread switches. Most concurrency errors appear with only a few thread switches [13]. By first eliminating these counterexamples, we forbid many other execution paths representing the same bug. TCBMC consists of four steps.

Step 1: Preprocessing. Complex program statements are not always executed atomically. However, if there is at most one occurrence of a global variable in a statement, context switches during the execution of the complex statement obviously cannot introduce concurrency-related errors. In contrast to this, context switches in statements that have more than one occurrence of global variables can introduce concurrency bugs. To model such context switches, we split statements with more than one reference to a global variable. This is done like in a compiler, where complex statements are broken down into simple instructions. For example, consider the statement $g_3 = g_1 + g_2;$, where g_1, g_2, g_3 are global variables. The statement is translated into $l_1 = g_1; \quad l_2 = g_2; \quad g_3 = l_1 + l_2;$, where l_1, l_2, l_3 are fresh local variables.

Step 2: Applying CBMC Separately on Each Thread. The next step is to unroll all loops, inline all function calls, and transform the code into *static single assignment form* (SSA form), where each variable is assigned only once. Hence, for each assignment to a variable, a new copy of this variable is created. Additionally, all variable names are prefixed with a thread identifier. E.g., for a global variable g (cf. Listing 2), copies "t_1g_1", "t_1g_2", etc. are created (cf. Listing 3). This second step is performed for each thread in isolation, as done in CBMC [9]. It yields a separate formula for each thread, not taking into account that an execution of a thread can be interrupted by another one, which may change global variables. This is dealt with by Step 4, where additional concurrency-related constraints are added. To illustrate Step 2, consider the simple program shown in Listing 2. After applying CBMC separately on each thread, we get a formula representing the two threads as illustrated in Listing 3.

Step 3: Generating Block Variables and Atomic Sections. During an execution, sequentially executed lines of code from one thread form a so-called *context switch block*. For each line l of thread t, a so-called *block variable* $block_t(l)$ is introduced. The value of the block variable encodes to which context switch block the line belongs. Lines with the same values for their block variables belong to the same context switch block, and the blocks are executed in increasing order. So, by choosing values for the block variables, the SMT solver establishes the scheduling of the threads. Potential values of the block variables for our example from Listing 2 are illustrated in Figure 2. The block variables have to satisfy the following constraints:

1) The first block value of each thread should be positive, i.e., $\forall t \in T : block_t(1) \geq 1$.
2) For all threads, the block variable values must increase monotonically w.r.t. line numbers within a thread, i.e., $\forall t \in T, l \in t : block_t(l) \leq block_t(l+1)$.
3) The values of the block variables are not allowed to change by one (at least one thread should be running in between), i.e., $\forall t \in T, l \in t : block_t(l) + 1 \neq block_t(l+1)$.
4) No block variable value must exceed a given bound n. This is enforced by $\forall t \in T : block_t(m) \leq n$, where m is the last line number of the respective thread.
5) Each block variable value can only occur in one thread, i.e., $\forall t \in T, l \in t : \forall t' \in T \setminus t, l' \in t' : block_t(l) \neq block_{t'}(l')$.

Note that these rules for the block variables differ from [21]. The authors in [21] only give a detailed description of how to encode the block variables for two threads. For extending this to the general case, they suggested to enforce a round robin scheme among the threads, or to introduce new variables that represent which thread runs in which context switch block. We tried both methods, but found out that our definition of the block variables is much more efficient.

To model an *atomic section* between two consecutive lines of code, it is enough to require that the block variables for these lines must be equal. For instance, to model an atomic section in thread 1 between line 2 and 3, we add the constraint $block_1(2) = block_1(3)$. By adding the constraint $t_1a_{2,3} \rightarrow block_1(2) = block_1(3)$, where $t_1a_{2,3}$ is a boolean variable, we can easily enable or disable atomic sections in our synthesis algorithm by setting $t_1a_{2,3}$ to true or false.

Step 4: Generating Constraints for Concurrency. We have to adjust the SSA statements of each thread, as constructed in Step 2, to capture context switches. A statement that reads a global variable has to distinguish if the global variable was last assigned in its own context switch block or in a previous one. In the former case, the local value of the global variable is up to date and can be used. In the latter case, another thread may have altered the global variable, and we need to take the value as assigned by the other thread. Hence, we create copies of the global variables for each block, storing the values of the global variables at the end of the block. The SSA statement can access the copies of the global variables when needed. This is illustrated in Fig. 2. In this example we have four context switch blocks, so we create four additional copies $g_{copy(1)}$ to $g_{copy(4)}$ for each global variable. At the end of each block, we store the value of the last assignment of the global variable in the respective copy.

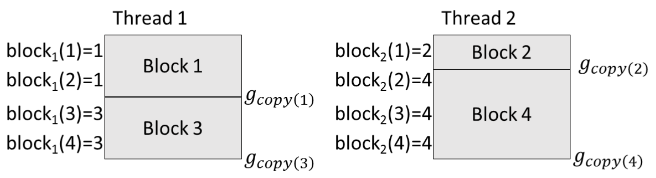

Fig. 2. Context Switch Blocks and Copy Variables.

Listing 4 SSA Constraints

```
procedure THREAD1
    if block($t_1x_1$) = block($t_1g_1$)
        $t_1x_1$ = $t_1g_1$;
    else
        b = block($t_1g_1$) - 1;
        $t_1x_1$ = $g_{copy(b)}$;
    $t_1x_2$ = $t_1x_1$ + 1;
    $t_1g_2$ = $t_1x_2$;
    $t_1x_3$ = $t_1x_2$ + 1;
procedure THREAD2
    if block($t_2y_1$) = block($t_2g_1$)
        $t_2y_1$ = $t_2g_1$;
    else
        b = block($t_2y_1$) - 1;
        $t_2y_1$ = $g_{copy(b)}$;
    $t_2y_2$ = $t_2y_1$ + 2;
    $t_2g_2$ = $t_2y_2$;
    $t_2y_3$ = $t_2y_2$ + 2;
```

Let us continue our example. After applying Step 4 to our SSA constraints from Listing 3 we get the final concurrency constraints shown in Listing 4, where $block(x)$ gives the context switch block in which the variable x was assigned. Note that we only have to change an SSA statement if it reads a global variable. In this case, we have to check if the local value is up to date, or if we must use the copy of the global variable from the previous context switch block.

Modeling Assumptions and Assertions: We extended the SMT encoding to also support assertions and assumptions, which are Boolean conditions in the input program P. A counterexample must satisfy all assumptions, but violate one assertion or sequential correctness. Hence, modeling assumptions and assertions in the SMT encoding is straightforward: For computing counterexamples, we add the constraints $\bigwedge_i \mathsf{assumption}_i \wedge \neg(\mathsf{seqSpec} \wedge \bigwedge_j \mathsf{assert}_j)$. When searching for valid runs, the negation ($\neg$) is omitted. Assumptions can, for example, be used to model `wait` statements.

C. Finding Atomic Sections with the FixSwitches Algorithm

We now turn to the first method to analyze counterexamples in order to infer a small but sufficient set of atomic sections.

Listing 5 FixSwitches Algorithm

```
procedure FixSwitches
  $A := \emptyset$
  while ce($A$) $\neq$ None do
    $\overline{s} := (s_1, s_2, \dots s_m) :=$ ce($A$)
    $(k_1, \dots, k_n) :=$ findSwitches($\overline{s}$)
    for $i = n \dots 1$ do
      if existsValidRun($(s_1, s_2, \dots s_{k_i})$) then
        $A := A \cup \{s_{k_i}\}$
        break
  return $A$
```

Listing 5 presents a method to compute atomic sections based on a heuristic to analyze counterexamples. As outlined in Fig. 1, it starts with an empty set of atomic sections A. In a loop, a new counterexample $\overline{s} = s_1, s_2, \dots s_m$ is computed that respects the atomic sections A that have already been found so far. If no such counterexample exists, then A must be sufficient and the algorithm terminates. Otherwise, the procedure `findSwitches` computes all non-mandatory context switches of the counterexample $\overline{s}$ in form of a sequence of indices k such that thread$(s_k) \neq$ thread(s_{k+1}) and $s_k \in V'$. Next, the algorithm analyzes the context switches of the counterexample in reverse order, i.e. starting with the last non-mandatory context switch k_n. The procedure `existsValidRun` now checks whether it is possible to extend the incomplete execution $s_1, s_2, \dots s_{k_n}$ to a complete one that is correct and does not have a context switch at k_n. If this is not the case, the program cannot be fixed just by forbidding the context switch k_n; a concurrency problem must already exist in an earlier stage of the execution $\overline{s}$. Thus, we continue to analyze the previous context switch k_{n-1}. Eventually, we must find an index i such that `existsValidRun`$(s_1, \dots, s_{k_i})$ returns true, because if there are no more switches left in the prefix, then a sequential execution is possible. If `existsValidRun` returns true, we add an atomic section that forbids the context switch k_i (thus making the current counterexample infeasible), and look for a new counterexample.

The procedure `existsValidRun` can be implemented similar to ce, based on an SMT-solver call. In the SMT-solver query of `existsValidRun`, we cannot only assert the execution path of the prefix but also all variable values (taken from the satisfying assignment of the SMT-call in ce) in the different execution steps of the prefix. This renders `existsValidRun`-calls typically much cheaper than ce-calls in terms of computation time. The performance of the entire algorithm increases if we consider counterexamples with a small number of context switches first, and increase the maximum number of (non-mandatory) context switches incrementally. That is, only if no more counterexamples with one context switch exists, we search for counterexamples with two context switches, and so on.

Listing 6 illustrates how this algorithm works. According to the implicit sequential execution, the global variable h

Listing 6 Fix Switches Example

```
 1: int g; int h = 0;
 2: procedure THREAD1                 7:  procedure THREAD2
 3:   g = 0;            --s1-->       8:    g = 1;
 4:   if g = 0 then     <--s2--       9:    if g = 1 then
 5:     int tmp = h;
 6:     h = tmp + 1;    --s3-->      10:      int tmp = h;
                                     11:      h = tmp + 1;
```

Listing 7 Fix Switches Example (continued)

```
 1: int g; int h = 0;
 2: procedure THREAD1                        7:  procedure THREAD2
 3:   g = 0; block(3)=1   --s1-->            8:    g = 1;        block(8)=2
                                             9:    if g = 1 then block(9)=2
 4:   if g = 0 then                         10:      int tmp = h;
 5:     int tmp = h;                        11:      h = tmp + 1;
 6:     h = tmp + 1;
```

should be 2 after executing Thread1 and Thread2 in parallel. Suppose, we get the following counterexample: $\overline{s} = s_1, s_2, s_3$, where $s_1 = \{3, 8\}$, $s_2 = \{9, 4\}$, and $s_3 = \{6, 10\}$. The last switch s_3 is a mandatory context switch. So in order to get rid of the counterexample, we can either forbid s_1 or s_2. First we investigate, whether switch s_2 is the bad switch. Therefore, we fix the execution until s_2. So first one line of thread 1 has to be executed (block(3)=1) and then two lines of thread 2 (block(8)=block(9)=2), see Listing 7. Now the algorithm checks whether it is possible to extend this incomplete execution to a complete correct one. Since this is not the case, s_2 is innocent, and the real problem lies in s_1. In the next step, the algorithm forbids s_1 by inserting an atomic section between line 3 and line 4.

D. *Finding Atomic Sections with the* AtomConstr *Algorithm*

The *Atomicity Constraint Algorithm* (AtomConstr), shown in Listing 8, is inspired by [24]. While FixSwitches added atomic sections to the set A in each iteration, AtomConstr only adds candidates for atomic sections to a set of sets $\mathcal{A}$. Initially, $\mathcal{A}$ is empty. The algorithm searches in a loop for counterexamples $\overline{s} = s_1, s_2, \dots s_m$ that respect $\mathcal{A}$ and computes all thread switches $K = \{k_1, k_2, \dots k_n\}$ of $\overline{s}$. The set K represents all possible ways to eliminate the

Listing 8 AtomConstr Algorithm

```
procedure AtomicityConstraint
  $\mathcal{A} := \emptyset$
  while ce'($\mathcal{A}$) $\neq$ None do
    $\overline{s} := (s_1, s_2, \dots s_m) :=$ ce'($A$)
    $K := \{k_1, \dots, k_n\} :=$ findSwitches($\overline{s}$)
    $\mathcal{A} := \mathcal{A} \cup \{K\}$
  return hittingSet($\mathcal{A}$)
```

TABLE I

Experimental Results. *The column #Lines gives the lines of code after preprocessing with FoREnSiC. All other columns give total execution times in seconds. We used a timeout (t.o.) of 5400 seconds. The numbers in brackets give the number of iterations in the refinement loop of Fig. 1.*

		integer arithmetic		uninterpreted func.	
	#Lines	AtomConstr	FixSwitches	AtomConstr	FixSwitches
RSA	23	–	–	1.5(2)	1.26(4)
`linEq` _2t 1	38	0.7(2)	0.9(2)	0.6(2)	0.6(2)
`linEq` _2t 2	55	43(4)	42(4)	1.7(4)	1.8(4)
`linEq` _2t 3	70	550(6)	623(6)	3.2(6)	4.8(6)
`linEq` _2t 4	87	4882(8)	5320(8)	6.9(8)	8.7(8)
`linEq` _2t 6	121	t.o.	t.o.	21(12)	17(12)
`linEq` _2t 8	155	t.o.	t.o.	44(16)	42(16)
`linEq` _2t 10	189	t.o.	t.o.	71(20)	86(20)
`linEq` _2t 12	223	t.o.	t.o.	117(24)	129(24)
`linEq` _2t 14	257	t.o.	t.o.	186(28)	169(28)
`linEq` _3t 1	52	25(3)	26(3)	2.3(3)	2.1(3)
`linEq` _3t 2	76	t.o.	t.o.	8.2(6)	7.8(6)
`linEq` _3t 3	97	t.o.	t.o.	18(9)	18(9)
`linEq` _3t 4	121	t.o.	t.o.	42(12)	38(12)
`linEq` _3t 6	169	t.o.	t.o.	113(18)	106(18)
`linEq` _3t 8	218	t.o.	t.o.	247(24)	258(24)
`linEq` _3t 10	265	t.o.	t.o.	398(30)	378(30)
`linEq` _4t 1	66	t.o.	t.o.	7(4)	3.9(4)
`linEq` _4t 2	97	t.o.	t.o.	28(8)	38(8)
`linEq` _4t 3	124	t.o.	t.o.	89(12)	90(12)
`linEq` _4t 4	155	t.o.	t.o.	150(16)	169(16)
`linEq` _4t 6	217	t.o.	t.o.	485(24)	506(24)
`VecPrime` 2	157	173(836)	53(108)	2.9(16)	3.1(16)
`VecPrime` 3	221	471(942)	190(162)	11(24)	12(24)
`VecPrime` 4	290	2018(1018)	519(2016)	66(32)	69(32)
`VecPrime` 5	359	t.o.	1356(2070)	627(40)	514(40)
IIO	60	1.1(9)	1.3(9)	0.9(9)	1.1(9)
CVE	150	11(21)	13(21)	4.1(12)	5.8(12)
TG3	133	17(74)	21(74)	9.8(74)	13(74)

counterexample $\overline{s}$: At least one of the switches from K must be forbidden by an atomic section to make $\overline{s}$ unfeasible. In the next step, we add K to $\mathcal{A}$. The set $\mathcal{A}$ consists of sets of atomic sections candidates and from each set, at least one of the atomic section must be active to forbid the corresponding counterexample. So $\mathcal{A}$ represents a CNF formula.

A hitting set for $\mathcal{A}$ is a set A that shares at least one common element with every set in $\mathcal{A}$. If no more counterexample exists, the minimal hitting set of $\mathcal{A}$ represents a global minimum of atomic sections. One efficient way to compute a minimal hitting set is described in [22].

V. Experimental Results

We have evaluated our approach experimentally, using a prototype implementation for concurrent C programs, called Atoss. It uses the front-end of the FoREnSiC [2] tool, which in turn uses gcc to parse the input C files. We have added a new back-end to FoREnSiC to create the SMT queries that we submit to the Z3 solver. The models returned by Z3 are the counterexamples that Atoss analyzes to create a refined SMT query, until Z3 returns UNSAT and we have found a solution. In addition to the illustrative example presented in Section III, we used two parameterized benchmarks called `linEq` and `VecPrime`, which can also be solved with integer arithmetic. This enables us to rate the effects of our abstraction with uninterpreted functions. To show that our approach is also applicable to real-world problems, we also ran Atoss on three bugs in Linux kernel modules. Our prototype implementation, all benchmarks, as well as scripts to run them are available for evaluation at http://www.iaik.tugraz.at/content/research/design_verification/atoss/.

Our experimental results are summarized in Table I. We show execution times to synthesize synchronization for each of the benchmarks, using our two different algorithms (FixSwitches and AtomConstr), comparing abstraction with uninterpreted functions and normal integer arithmetic.

The RSA example has already been explained in Section III. This benchmark can only be solved by abstraction with uninterpreted functions, as the complex arithmetic functions involved are beyond the capabilities of state-of-the-art `QF_LIA` solvers. FixSwitches finds the atomic sections one would naturally expect (lines 6–7 and 13–14; see Section III). Interestingly, AtomConstr computes a different solution of the same size: it suggests to make the lines 5–6 and 12–13 atomic. This is also correct because if each thread decides on the merge right after being finished, only the second thread to finish can enter the `if` to do the merge.

The `linEq` benchmark is based on the idea of checking whether a given n-tuple satisfies a given linear equation with n variables. Multiplications of the equation's coefficients with the elements of the n-tuple is distributed over multiple threads. This example is scalable w.r.t. two different parameters: the number of threads, and the size of n. The naming convention in Table I is as follows: "`linEq` _3t 4" has 3 threads and $n = 4$. We can see from Table I that using uninterpreted functions for abstraction significantly speeds up the synthesis process, and even enables synthesis for many benchmarks that would timeout otherwise. Concerning scalability, it should be noted that each of these benchmarks contains n times the number of threads potential race conditions. In real-world examples, we usually expect a much lower number of potential concurrency issues. These benchmarks were specifically designed to challenge the scalability of our approach.

The idea of `VecPrime` benchmark is that there is a vector of numbers, and we want to count the contained prime numbers. One thread starts counting from the "left" side of the vector, the other one starts from the "right" side. Every number that has been taken into account is set to 0. This way it is ensured that no element is counted twice.[1]

We also applied our tool to three real world examples. The first one (`linux_iio`) is based on a bug[2] found in the industrial I/O subsystem (IIO) of the linux kernel. IIO

[1]We assume that the check `isPrime(0)` is significantly faster than other calls to `isPrime`. Thus, it hardly matters for efficiency that both threads go through the entire vector, for simplicity.

[2]http://git.io/JjCEXg

polls hardware-devices for triggers, to notify consumers of events, e.g. that new data is available. A global variable counts the number of running threads. The race condition occurs, if several trigger-consumer modify this variable simultaneously.

The second example, the `CVE-2014-0196` benchmark is based on a bug[3] in a Linux kernel module, which has only been discovered very recently. Slightly simplified, a race condition could lead to an erroneous value in a variable that counts how much space remains in a buffer, which can subsequently result in a buffer overflow. This can lead to memory corruption, and exploits have been published that allow crashing the system or gaining root access. We have fed the relevant part of the kernel module's code (150 lines of code) to Atoss.

Finally, the last example (`linux_tg3`) is based on a bug[4] found in the Broadcom Tigon3 (TG3) ethernet driver. In the retrieval function for hardware statistics, the driver retrieves the statistics from the device and stores it into a buffer. Since the tg3 driver updates the hardware statistics in a non-atomic way, the state of the statistics can get inconsistent.

For all three real-world examples, we did not have to add any specification, but just relied on the implicit specification given by serializability. Within just a few seconds, Atoss was able to find a fix. For CVE and TG3, the computed fix matches the "official" fix that has been made by the kernel community. For IIO our tool found a slightly different fix.

When comparing FixSwitches with AtomConstr, there is no clear winner. Each algorithm is faster for some examples. It should be noted that both algorithms always found a globally minimal set of atomic sections for all our benchmarks.

VI. Conclusion

We have presented a new approach to synthesis of synchronization for concurrent programs. Using uninterpreted functions, we are able to efficiently abstract details of the program that are irrelevant for concurrency issues. We have shown experimentally that this abstraction is more efficient than just using integer arithmetic without any abstraction. Also, we are able to handle benchmarks that would not have been feasible at all, using integer arithmetic.

Moreover, we have demonstrated that this approach can be applied to real-world concurrency issues, such as race conditions in kernel modules. In particular, the applicability of our approach is supported even further by a very low entry barrier. We do not require designers to write a formal specification. They can simply run our tool on their code as it is, and still detect and fix concurrency issues.

Due to this encouraging results, we plan to do future work on several improvements and optimizations. First, we would like to add support for commutative and associative (yet still uninterpreted) functions, to improve the abstraction/expressibility trade-off. This should lead to a performance boost for benchmarks where the order of operations is not relevant for the final result. Second, we note that in practical examples, concurrency bugs are usually limited to a few lines of code only. Thus, we would like to be able to automatically disregard program parts that do not contain any concurrency bugs, by abstracting them with uninterpreted functions. This should improve scalability so that we could deal more easily with even larger real-world examples.

[3]https://bugzilla.redhat.com/show_bug.cgi?id=1094232

[4]http://git.io/7wWrKw

References

[1] L. Bachmair, I. V. Ramakrishnan, A. Tiwari, and L. Vigneron. Congruence closure modulo associativity and commutativity. In *FroCoS'00*, LNCS 1794, 2000.

[2] R. Bloem, R. Drechsler, G. Fey, A. Finder, G. Hofferek, R. Könighofer, J. Raik, U. Repinski, and A. Sülflow. FoREnSiC - an automatic debugging environment for C programs. In *HVC'12*, LNCS 7857. Springer, 2012.

[3] R. E. Bryant, S. M. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Trans. Comput. Log.*, 2(1):93–134, 2001.

[4] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV'94*, LNCS 818. Springer, 1994.

[5] P. Cerný, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *CAV'13*, LNCS 8044. Springer, 2013.

[6] S. Cherem, T. M. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI'08*. ACM, 2008.

[7] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*. Springer, 1981.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[9] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS'04*, LNCS 2988. Springer, 2004.

[10] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs'09*, LNCS 5674. Springer, 2009.

[11] L. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In *ICSE'11*. ACM, 2011.

[12] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, 2008.

[13] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS'03*. IEEE, 2003.

[14] M. K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. In *SPIN'08*, LNCS 5156. Springer, 2008.

[15] G. Hofferek and R. Bloem. Controller synthesis for pipelined circuits using uninterpreted functions. In *MEMOCODE'11*. IEEE, 2011.

[16] G. Hofferek, A. Gupta, B. Könighofer, J.-H. R. Jiang, and R. Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In *FMCAD'13*. IEEE, 2013.

[17] U. Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *AAAI'04*. AAAI Press / The MIT Press, 2004.

[18] V. Kahlon. Automatic lock insertion in concurrent programs. In *FMCAD'12*. IEEE, 2012.

[19] A. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[20] A. Pnueli, O. Strichman, and M. Siegel. The code validation tool CVT: Automatic verification of a compilation process. *STTT*, 2(2):192–201, 1998.

[21] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *CAV'05*, LNCS 3576. Springer, 2005.

[22] R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1), 1987.

[23] A. Solar-Lezama, C. G. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI'08*. ACM, 2008.

[24] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL'10*. ACM, 2010.