Hindawi
Wireless Communications and Mobile Computing
Volume 2018, Article ID 3193974, 8 pages
https://doi.org/10.1155/2018/3193974

Research Article

An SAT-Based Method to Multithreaded Program Verification for Mobile Crowdsourcing Networks

Long Zhang,[1] **Wanxia Qu**,[1] **Yinjia Huo**,[2] **Yang Guo**,[1] **and Sikun Li**[1]

[1] *College of Computer, National University of Defense Technology, Changsha, China*
[2] *The University of British Columbia, Vancouver, BC, Canada V6T 1Z4*

Correspondence should be addressed to Wanxia Qu; quwanxia@nudt.edu.cn

Received 23 September 2017; Revised 24 December 2017; Accepted 1 January 2018; Published 28 January 2018

Academic Editor: Edith Ngai

This paper focused on the safety verification of the multithreaded programs for mobile crowdsourcing networks. A novel algorithm was proposed to find a way to apply IC3, which is typically the fastest algorithm for SAT-based finite state model checking, in a very clever manner to solve the safety problem of multithreaded programs. By computing a series of overapproximation reachability, the safety properties can be verified by the SAT-based model checking algorithms. The results show that the new algorithm outperforms all the recently published works, especially on memory consumption (an advantage that comes from IC3).

1. Introduction

The mobile crowdsourcing network is a promising network architecture to perform tasks with human involvement and numerous mobile devices but suffers from security and privacy concerns [1, 2]. Pthread-style multithreaded programs play an important role in crowdsourcing computing [3, 4] and crowdsourcing sensing [5, 6] for supporting concurrent programming. Multithreaded programming used much existing system-level code such as device drivers, operating system, and distributed computing. The mobile crowdsourcing networks [7, 8] distribute tasks and collect the results and must make sure safely access for the shared data. Therefore, it is important to verify the safety properties for multithreaded programs.

In this paper, we consider the multithreaded programs with an unbounded number of threads. Each thread executes a finite, nonrecursive state machine. Mutexes, which can be expressed using Boolean variables, are used for synchronization in this kinds of multithreaded programs. The illegal access of mutexes will lead to safety problem. We assume all mutexes are shared variables. The safety problems is to verify if the illegal access of mutexes exists.

The safety property for multithreaded programs which was expressed as upward-closed sets of the target ("bad") states can be verified by reduction to the coverability problem of *well-structured transition systems* (WSTS) [9, 10]. WSTS are a very broad class of infinite-state systems, including thread transition system (TTS) [11], Petri nets and their monotonic extensions [12–16], broadcast protocols [17, 18], lossy channel systems [19] and context-free grammars [10]. Coverability is the base verification task for WSTS: the question is whether the system can reach an unsafe or illegal configuration among some subset of its (possibly unbounded number of) components. There have been several algorithms published for WSTS coverability problem [9–11, 20–22], but none perform as efficiently as finite state model checking.

The IC3 algorithm [23] is an SAT-based model checking algorithm and introduced as an efficient technique for safety properties verification of finite state systems, especially in hardware verification. It computes an inductive invariant by maintaining a sequence of overapproximation of reachability from initial states and strengthens them incrementally. An efficient implementation of the procedure shows good performance on hardware benchmarks [24].

This paper focuses on the multithreaded programs for mobile crowdsourcing networks. All multithreaded programs are pthread-style ANSI-C source code and transformed into TTS by using predicate abstraction [25, 26]. We introduce a novel, highly efficient algorithm for the coverability problem of TTS. The new algorithm is to find a way to apply conventional, finite state IC3, which is typically

the fastest algorithm for finite state model checking, in a very clever manner to solve the coverability problem. IC3 algorithm is a finite state model checking algorithm, and the original input is an FSM. We try to use the finite state model checker to solve the infinite-state systems. The bounded TTS is transformed into an FSM and described as the inputs format for IC3 engine. The significant contributions of this paper are as follows:

(1) Our approach requires very novel and intricate reasoning because IC3 produces a series of overapproximation reachability results. A novel algorithm which is based on IC3 engine is proposed to solve the coverability problem of TTS.

(2) We introduce new encoding techniques to make the verification of infinite-state systems possible by using finite state algorithms.

(3) We implement tool's combination, which is a good way to improve the total rate of successfully solved instances.

The experimental results show that our new algorithm outperforms all the recently published works, uses far less memory (an advantage that comes from IC3), and can solve more benchmarks successfully. The new method can solve 97.2% instances within 1 GB.

The rest of this paper is organized as follows. In Section 2, we review the related work. Section 3 presents necessary preliminaries used in this paper. In Section 4, we propose our new method based on IC3 and give more details of the implementation. Section 5 shows the experimental evaluation on multithreaded programs. Section 6 concludes this paper and discusses future works.

2. Related Works

A general decidability result showed that the coverability problem is decidable for WSTS [9], which backward-explore states starting from the target states. Bingham and Hu [20] proposed a new algorithm to compute fix-points over a series of finite state systems of increasing size. A new subclass of WSTS, named Nice Sliceable WSTS, was introduced. Starting from the target states, it computed the exact backward reachability by using finite state symbolic model checking [27] based on BDDs [28] to solve the coverability problem of NSW. Kaiser et al. [11, 22] introduced a new algorithm to solve the safety properties of multithreaded programs with an unbounded number of threads executing a finite state, nonrecursive procedure. By using many inexpensive uncoverability proofs, this new approach combined forward propagation under-approximations with backward propagation of overapproximations to the coverability problem in TTS. Inspired by the success of IC3 algorithm in finite state model checking, Kloos et al. [21] proposed an incremental, inductive procedure to check coverability of downward-finite WSTS, which contains Petri nets, broadcast protocols, and lossy channel systems. All those algorithms are based on the backward reachability and suffered from complex computational consumption. Esparza et al. [29] introduced an incomplete but empirically efficient solution to the coverability problem. The new approach was based on classical Petri nets analysis techniques, the marking equation and traps [30, 31], and utilized an SMT solver to implement the constraint approach. Inspired by Esparza's work, Athanasiou et al. [32] introduced an approximate coverability method by using thread state equations and implemented it in a tool named TSE. TSE is very capable on Boolean programs but theoretically incomplete.

3. Preliminaries

3.1. Nicely Sliceable WSTS

Definition 1 (well-quasi-ordering). A well-quasi-ordering (wqo) is a reflexive and transitive binary relation $\preceq$ over set X, and for every infinite sequence $x_0, x_1, x_2, \ldots$ of elements from X, there exists $i < j$ such that $x_i \preceq x_j$.

For $Y \subseteq X$, the upward-closure of Y is the set $\uparrow Y = \{x \mid \exists y \in Y,\ y \preceq x\}$. A basis of an upward-closed set Y is a set $B \subseteq Y$ such that $Y = \uparrow B$. A set U is said to be $\preceq$-upward-closed (or simply upward-closed if $\preceq$ is clear from the context) if $U = \uparrow U$. It is known that if $\preceq$ is a *wqo*, then any $\preceq$-upward-closed set has a unique finite basis B such that for all $x, y \in B$ we have $x \not\preceq y$ and $y \not\preceq x$ [33]. Given upward-closed U, we let $basis(U)$ denote the unique finite basis of U. Moreover, it is known that any infinite increasing sequence $S_0 \subseteq S_1 \subseteq S_2 \subseteq \cdots$ of upward-closed sets eventually stabilizes; that is, there exists $k \in \mathbb{N}$ such that $S_k = S_{k+1} = S_{k+2} = \cdots$.

Definition 2 (discrete wqo). A wqo is a discrete wqo (dwqo) over X if for all $x \in X$ there exists $k \in \mathbb{N}$ such that for any sequence $x_0 \prec x_1 \prec \cdots \prec x_l = x$, we have $l \leq k$. The weight function $w : X \to \mathbb{N}$ maps each x to the minimum such as k. For the $\preceq$-upward-closed set U, the base weight of U is $bw(U) = \max\{w(x) \mid x \in basis(U)\}$.

The weight function $w(x)$ slices the state space into a countable number of finite sets $S_0, S_1, S_2, \ldots$, where $S_i = \{x \in S \mid w(x) = i\}$. This property allows for finite state model checking techniques to be used to the reachability for each weighted bounded S_i.

Definition 3 (nicely sliceable well-structured transition systems). A nicely sliceable well-structured transition system (NSW) $M = (S, \to, \preceq)$ is a transition system equipped with a dwqo on its states that satisfies the following properties:

(1) S is the (possibly infinite) state space.

(2) $\to \subseteq S \times S$ is transition relation.

(3) $\preceq$ is a dwqo over S.

(4) For all $x, x', y \in S$, if $x \to x'$ and $x \preceq y$, there exists y' such that $y \to y'$ and $x' \preceq y'$.

(5) Weight-respecting: for all $x, x', y \in S$, $w(x') - w(x) = w(y') - w(y)$.

(6) δ-deflatable: for $\delta \in \mathbb{N}$ if whenever $x \to x'$ and $z \preceq x'$, there exists y and y' such that the following properties hold: (1) $y \preceq x$, (2) $y \to y'$, (3) $z \preceq y'$, (4) $w(y) \leq w(z) + \delta$, and (5) $w(y') \leq w(z) + \delta$.

3.2. Thread Transition Systems. Thread transition systems (TTS) are motivated by the verification task of multithread asynchronous programs, which is the subset of NSW. Let L and S be finite sets for *local* and *shared* states, respectively. The elements of $T = S \times L$ are called *thread states*.

Definition 4 (thread transition system). A thread transition system (TTS) is a pair (T, R), where $R \subseteq T \times T$ is a binary relation on T, partitioned into $R = \mapsto \cup \hookrightarrow$.

Let $V = \bigcup_{n=1}^{\infty}(S \times L^n)$. The elements of V are called *states*. We write them in the form $(s \mid l_1, l_2, \ldots, l_n)$. A TTS gives rise to a transition system $M = (V, \rightarrowtail)$ with

$$(s \mid l_1, l_2, \ldots, l_n) \rightarrowtail \left(s' \mid l'_1, l'_2, \ldots, l'_{n'}\right), \tag{1}$$

if one of the following conditions holds.

Thread Transitions. $n' = n$ and there exists $(s, l) \mapsto (s', l') \in R$ and i such that $l_i = l$, $l'_i = l'$, and, for all $j \neq i$, $l'_j = l_j$.

Spawn Transitions. $n' = n+1$ and there exists $(s, l) \hookrightarrow (s', l') \in R$ and i such that $l_i = l$, $l'_{n'} = l'$, and for all $j < n'$, $l'_j = l_j$.

Let $I_l \subseteq L$ be a set of initial local states and $I_s \subseteq S$ be a set of initial shared states. We define the set of initial states to be $I = I_s \times \bigcup_{n=1}^{\infty} I_l^n$. An execution of the transition system M is a finite or infinite sequence of states in V whose adjacent states are related by $\rightarrowtail$, which started at initial states. A state is reachable if it appears in some execution.

In order to state the coverability problem, define the *cover* relation $\preceq$ over V as $(s \mid l_1, l_2, \ldots, l_n) \preceq (s' \mid l'_1, l'_2, \ldots, l'_{n'})$ if $s = s'$ and $[l_1, l_n, \ldots, l_n] \subseteq [l'_1, l'_2, \ldots, l'_{n'}]$, where $[\cdot]$ denotes a *multiset*.

Give target states $v_F \in V$, if v_F is coverable; that is, does there exists a path in M leading to a state v that covers v_F: $v \succeq v_F$? The safety property is described as the upward-closed set of v_F and converts into the coverability analysis problem.

A *cover* relation $\preceq$ is neither symmetric nor antisymmetric, thus a quasi-order, and in fact a well-quasi-order (wqo) on V: any infinite sequence $v_1, v_2, \ldots$ of elements from V contains an increasing pair $v_i \preceq v_j$ with $i < j$. It is easy to see that $(M, \preceq)$ fulfills the definition of WSTS. A TTS with standard thread and spawn transition can be expressed as plain Petri nets [22] and is the subset class of NSW [20].

4. Multithreaded Programs Safety Verification

In this section, we introduce a new method for the safety verification of multithread programs. The input source code is translated into TTS by using SatAbs [34]. Then, we propose a novel TTS coverability analysis algorithm to verify the safety properties. Finally, the implementation details are described.

4.1. The Input Languages. Most popular programming languages such as Java and C/C++ embrace concurrent programming via their *pthread* or *thread* class APIs, respectively. In this paper, we focus on the pthread-style multithreaded ANSI-C programs. ANSI-C is one of the most popular programming languages for safety critical embedded software. The mobile crowdsourcing networks contains most embedded devices which are based on multithreaded programs to support the crowdsourcing computing and communications. SatAbs is an SAT-based model checker by predicate abstraction, and can be used to model the ANSI-C programs into TTS format. We follow the introduction from the SatAbs website (http://www.cprover.org/satabs/) to transform the ANSI-C programs into TTS. All ANSI-C programs can be translated into Boolean programs by SatAbs, completely. The safety properties can be reserved during the formalization process.

4.2. IC3-Based Thread Transition System Coverability Analysis Algorithm. IC3 is SAT-based and computes inductive overapproximations of reachable sets. Let I and P be initial states and the property states, respectively. Also let T denote the transition relation over the current and the next states. IC3 maintains a trace: $[R_0, R_1, \ldots, R_N]$. The first element R_0 is the initial states. For $i > 0$, R_i is a set of clauses that AND-ed together and represent an overapproximation of the states reachable from the initial states in k steps or less. $R_i \rightarrow R_{i+1}$, and the clauses R_{i+1} are a subset of R_i, except for $i = 0$. The IC3 algorithm will terminate if a counterexample is found or an inductive proof R_N is got.

This section develops our new algorithm, TTSCov, which is based on the IC3 algorithm. For a target set v_F and $i \in \mathbb{N}$, $wl(v_F, i)$ presents the set of weight limited v_F by i. From the base weight of v_F, the algorithm computes the overapproximation for the backward reachable set O_i. O_i is an inductive overapproximation of the states from v_F which is reachable along a path that never exceeds weight i. We use IC3, an SAT-based finite state model checker, to compute this weight limited and inductive overapproximation of v_F.

As shown in Algorithm 1, the input is a TTS M, a set of initial states I, and an $\preceq$-upward-closed set of target states v_F. The variable i is the current weight boundary, which is initially the base weight of v_F and increases by 1 each loop iteration. O_i is an overapproximation of $br(v_F, i)$, which initially is set as $wl(v_F, bw(v_F))$. O_i is an overapproximation of O_{i-1} bounded by the weight i, which is computed by IC3 engine. If O_i intersected with the initial states I, the counterexample was found, and the algorithm terminated. In line (6), we check if O_i and O_{i-1} are equal, if not, the variable n was assigned as the current i. If the condition of line (6) fails δ times consecutively, we have $O_{n+\delta} = O_{n+\delta-1} = \cdots = O_n$, and thus the verification is successful.

Actually, the condition in line (8) can be replaced by $O_i \not\equiv O_{i-1}$, and the algorithm works well under the syntactically equal check. But the syntactic checking is more efficient, as the IC3 algorithm builds the frame incrementally, and the clause in O_i is the subset in O_{i-1}. In order to take this feature, a Boolean variable can be used to check if new clauses add to O_i when computing the overapproximation at line (3). This speeds up the algorithm much more.

The main routine in Algorithm 1 is the while-loop. For each loop, the IC3 engine computes the overapproximation by using the SAT solver. The algorithm terminates when it finds a counterexample at line (5) or proves safety at line (13).

Intput:
$M = (S, \rightarrow, \preceq)$: a TTS model to describe a multi-threaded program
$I \subseteq S$: the inital states
δ: a constant which depend on the input model M
v_F: the target states, which is described as an upward-closed set
Output:
safe or *unsafe*
(1) $i := bw(v_F)$, $n := i$, $O_{i-1} := wl(v_F, i)$ // initialization
(2) **while** $n \geq i - \delta$ **do**
(3) $O_i := IC3(O_{i-1}, I, i)$
(4) **if** $O_i \cap I \neq \emptyset$ **then**
(5) print *CEX*
(6) RETURN *unsafe*
(7) **end if**
(8) **If** $O_i \neq O_{i-1}$ **then**
(9) $n := i$
(10) **end if**
(11) $i := i + 1$
(12) **end while**
(13) RETURN *safe*

Algorithm 1: TTSCov.

4.3. Implementation. A TTS with thread and spawn transitions is expressive as a plain Petri net. Zhang et al. [35] introduced a method to cut off a Petri net into a finite state machine (FSM). Inspired by Zhang's work, this section introduces the details of how to bound the TTS into FSM.

The TTS Format. The input multithreaded programs model are encoded in the TTS format (http://www.cprover.org/bfc/). Each *shared* or *local* is mapped to a *shared/local* variable. Just one *shared* variable can be assigned to "1," and all *local* variables could be assigned to arbitrary natural number. A transition is a thread transition or spawn transition, which described how the thread state changed.

The FSM Format: AIGER. AIGER is a format, library and set of utilities for And-Inverter Graphs (AIGs) (http://fmv.jku.at/aiger/). AIGs are a good way to describe a FSM and can be translated into a propositional logic for a SAT solver. The bounded TTS is encoded as an AIG model, where each *shared* and *local* state corresponds to state variables. Extra input variables are introduced to select which rule to be fired, and then update the state variables' value to set up the transition relations equally.

Shared and Local Variables. An $|S|$-bit vector was used to encode all *shared* variables, as just one shared variable was assigned with "1" at the same time. For each *local* variable, the unary encoding was used to encode the natural numbers n, as literature [35] shows that one-hot encoding is one possible unary encoding. As shown in Table 1, thermometer encoding is another unary encoding and performs well when using the incremental SAT solvers. In this paper, thermometer encoding is used to encode the *local* variables.

A full adder is used to bound the total thread number, and the logic is the same as described in [35]. The total thread number is the sum of all *local* variables' value. As the thermometer encoding technique is used to encode the local variables, the structure information is also added as the constraint to the AIG model.

Table 1: Encoding: binary, one-hot, and thermometer.

n	Binary	One-hot	Thermometer
0	000	00000001	00000001
1	001	00000010	00000011
2	010	00000100	00000111
3	011	00001000	00001111
4	100	00010000	00011111
5	101	00100000	00111111
6	110	01000000	01111111
7	111	10000000	11111111

5. Experimental Evaluation

We have implemented our algorithm in a tool named TTSCov. TTSCov is implemented with C++, and all input instances are encoded in TTS format. Petri nets tools are used by converting TTS instances into MIST format (https://github.com/pierreganty/mist). Most crowdsourcing programs are described in pthread-style multithreaded ANSI-C. SatAbs is the front-end of TTSCov, which translates the input ANSI-C programs in to TTS.

To measure TTSCov's performance, we compare with the state-of-the-art tools: MIST, IIC [21], BFC [11], Petrinizer [29], and TSE [32]. All experiments are performed on an Intel 3.4 GHz Intel, and 16 GB of memory, running Linux OS in 64-bit. The CPU time is limited to 1 hour, and memory to 10 GB.

5.1. Benchmarks. We collect 178 Petri nets examples from the Petrinizer repository (https://github.com/cryptica/pnerf), in

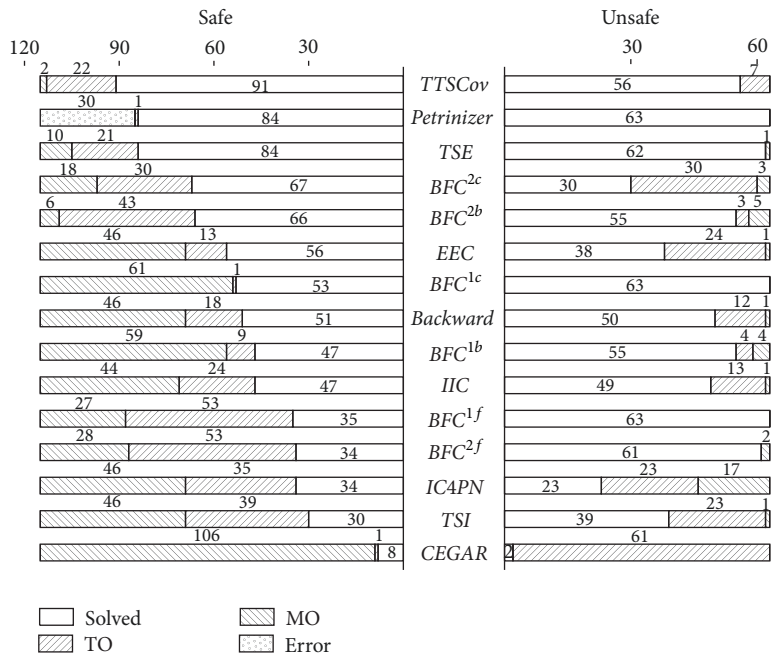

Figure 1: Total instances. All 178 instances are separated into safe and unsafe by the real results. There are 115 safe instances, and 63 unsafe instances. TTSCov compare with the state-of-the-art tools: MIST, IIC, BFC, Petrinizer, and TSE. There are two versions of BFC tool, and for each version, the tool has three modes. BFC^{2c} means the BFC v2.0 run in concurrent mode. f and b respect forward and backward, respectively. MIST toolkit implemented five algorithms: *Backward*, *EEC*, *TSI*, *IC4PN*, and *CEGAR*.

which 115 instances are safe and 63 instances are unsafe. All examples are organized into five suites. The first suite is a collection of plain Petri nets from the MIST toolkit. This suit contains 23 Petri nets (17 instances are safe, and the rest are unsafe) and 6 bounded Petri nets which are all safe. The second suite comes from the provenance analysis of message in a medical system, which contains 12 safe instances. The third suite also comes from the provenance analysis of message in bug tracking application, which contains 40 safe instances and 1 unsafe instance. The fourth suite contains 46 instances that are used to evaluate the BFC tool(http://www.cprover.org/bfc). Those instances are generated from concurrent C program in TTS format. They are mostly unsafe, and just 2 instances are safe. The fifth suite contains 50 instances that comes from the Erlang verification tool called Soter [36], and those examples can be found on Soter's Website (http://mjolnir.cs.ox.ac.uk/soter). Out of 50 instances in this suite, 38 are safe. This suite contains the largest example in the collection, with 66,950 places and 213,635 transitions.

5.2. Rate of Success on All Instances. We run two different version of BFC, named BFC v1.0 and BFC v2.0, respectively. The BFC tool has 3 modes: backward, forward, and concurrent. In concurrent mode, two threads are used to run backward and forward parallel. But we find there is a potential bug in BFC v2.0 with concurrent mode. The tool got stuck in concurrent mode when switching the thread. We run 10 times in concurrent mode and show the median results in Figure 1. Petrinizer is an SMT-based tool, which has 8 parameters, and the tool may return wrong answer for some examples. If Petrinizer returns wrong result in one configuration, we say the result is wrong. All five algorithms in MIST toolkit were compared.

Figure 1 shows that TTSCov performs better than the other complete tools, which solve 147 instances in total the same as Petrinizer, which is incomplete. For safe instances, TTSCov solves 91 out of 115, and 22 instances timeout. Most importantly, there are just 2 instances over the memory limit, which support the IC3 less memory usage. Petrinzier solves 84 instances, and 1 timeout. But 30 instances return unsafe results for those safe instances. Out of 30 wrong results examples, 19 instances return wrong answer with all 8 configurations, and the others are partly wrong. TSE solves 84 instances, 10 out of memory and 21 timeout. BFC^{2c} solves 67 examples in total, whereas 30 examples time out and 18 instances are out of memory. BFC^{2b} performs same as BFC^{2c}, in which 66 instances are solved, but with more out of time instances. BFC^{1c} solves 53 out of 115 safe instances, and 61 examples out of memory limit, with just one timeout. BFC^{1b} solves 47 safe instances, and 9 out of time limit and 59 out of memory limit. The forward algorithm performs worst in both BFC v1.0 and BFC v2.0, where 34 instances and 35 instances are solved by BFC^{2f} and BFC^{1f}, respectively. There are 53 instances of timeout for both tools. The BFC^{2f} has 28 examples out of memory, and the BFC^{1f} runs out of memory limit for 27 examples. IIC solves 47 instances, 24 instances timeout and 44 instances out of memory. For five algorithms in MIST toolkit, EEC solves 56 instances, 13 instances timeout, and 46 out of memory. Backward algorithm performs the same as EEC, 5 more instances

Table 2: Petrinizer and TSE combine with the other tools. For each suite, we focus on the instance that Petrinizer or TSE are unsolved, but the combined tool is solved, then present the total number of solved instances. All data are under taking the sum of the instances that Petrinizer or TSE can solve and the disjoint instances that the other tools can solve.

	Petri (29)	Medical (12)	Bug-tr (41)	Wa-kr (46)	Soter (50)	Total (178)
Petrinizer	21	4	33	46	43	147
Petrinizer + MIST	29	12	33	46	47	167
Petrinizer + BFC	27	4	33	46	49	159
Petrinizer + IIC	29	9	33	46	47	164
Petrinizer + TTSCov	29	12	41	46	49	177
TSE	19	4	33	46	44	146
TSE + MIST	29	12	33	46	47	167
TSE + BFC	27	4	33	46	49	159
TSE + IIC	29	9	33	46	47	164
TSE + TTSCov	29	12	41	46	49	177

timeout. TSI and IC4PN solve about 30 safe instances, but CEGAR just solves 8.

For unsafe instances, TTSCov solves 56 examples out of 63, no out of memory case, but 7 timeout. Petrinizer and BFC perform well in those suite cases. Petrinizer, BFC^{1c}, and BFC^{1f} solve all 63 unsafe instances. TSE solves 62 instances, and just one out of memory. BFC^{1b} has 4 timeout and 4 out of memory, respectively. BFC^{2b} has 8 instances out of time or memory limitation, and BFC^{2f} just has 2 instances out of memory. BFC^{2c} performs worse, where 30 instances are solved, because of the potential bugs in concurrent mode. For some unsafe instances in the fourth suite, the forward thread has found the counterexample, but got stuck when switching the thread until timeout. What is more, there are 3 examples out of memory when running BFC v2.0 in concurrent mode. IIC and Backward performs almost the same, which solve about 50 instances, 1 out of memory and the rest are timeout. EEC, TSI, IC4PN, and CEGAR are not good on unsafe instances, especially CEGAR, which only solves 2 unsafe instances.

In brief, TTSCov performs well both for safe and unsafe instances, especially in memory usage. Petrinizer performs well in time and memory usage for all instances, but reports wrong answer for safe instances. TSE performs nearly the same as TTSCov, but incomplete the same as Petrinizer. BFC v1.0 and v2.0 are good at unsafe cases, excluding the potential bugs in BFC^{2c}. IIC and MIST perform the same, but take more memory usage.

5.3. Tools Combination. Petrinizer and TSE are incomplete, but perform excellently on time and memory usage. We combine the Petrinizer and TSE with the other tools, and the total solved instances number is shown in Table 2.

Petrinizer works out 147 instances alone, but 30 instances return wrong or partly wrong result. For BFC, we compared the BFC v1.0 and BFC v2.0 in all three modes and then chose the best one. MIST stands for *Backward* algorithm, which performs the best of five algorithms in the MIST toolkit. When combining BFC with Petrinizer, 159 instances are solved. IIC and MIST solve 164 and 167, respectively, when working together with Petrinizer. Our tool TTSCov can solve 177 instances when combined with Petrinizer.

TSE solves 146 instances in total. When combined with the other tools, the total solved instances number is the same as Petrinizer. More importantly, TTSCov can solve all collected instances when combined with Petrinizer or TSE, except one instance from the Soter suite, which all tools can not deal with.

5.4. Memory Usage Evaluation. To show the memory usage of TTSCov, we compare with MIST, IIC, and BFC. Figure 2 shows that TTSCov is an efficient tool in memory usage, due to the use of IC3 as the back-end engine. TTSCov solves nearly 97.3% instances within 1 GB memory. About two-thirds instances can be solved within 2 GB for all tools, but TTSCov and BFC perform better than MIST and IIC in large instances. We find a bug when running BFC v2.0 for some instances from the forth instance suite. The tool has a segmentation fault for two instances, and we have get the bug confirmation from the author. The segmentation fault instances are marked as out of memory. Petrinizer and TSE are indeed efficient for memory usage, but incomplete. Therefore, we do not compare with those. TTSCov is based on the IC3 engine, which solves the verification problem without unrolling the transition relations. This is the main reason for why TTSCov performs well in memory usage. In conclusion, TTSCov is an efficient tool in memory usage, especially for huge instances.

6. Conclusion and Future Works

This paper introduce an IC3-based algorithm to verify the safety properties of multithreaded programs in mobile crowdsourcing networks. The pthread-style multithreaded program is modeled as a TTS. Then the state-of-the-art SAT-based model checking algorithm is used to verify the safety properties, by computing a series of overapproximation reachability with IC3. The results show that our new approach can solve more instances compared favorably against several recently published approaches. Due to using IC3 as the back-end engine, our method is significant for its lower memory consumption. Tools combination is a good direction to

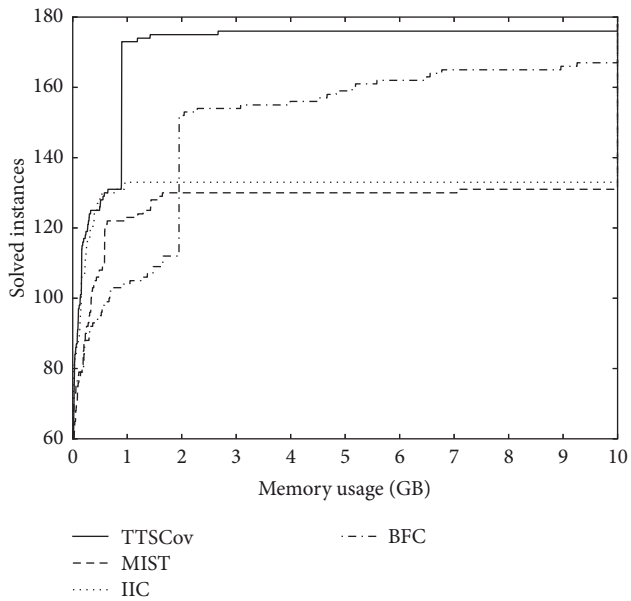

Figure 2: Comparison of TTSCov with MIST, IIC, and BFC for memory usage. BFC stands for BFC^{2b}. We use two versions BFC in all three modes and find that the BFC^{2b} performs the best in memory usage test. *Backward* algorithm is the best of five algorithms for memory usage in MIST toolkit, so MIST represented the *Backward* algorithm. For all timeout instances, we also show the memory usage in this figure.

solve the multithreaded programs for more complex mobile crowdsourcing networks. Parallel programming will be a good way to speed up the TTSCov algorithm.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The authors would like to acknowledge that this work was supported by the National Natural Science Foundation of China (Grant no. 61133007). They also thank Carl Kwan for helpful and detailed comments and suggestions.

References

[1] K. Yang, K. Zhang, J. Ren, and X. Shen, "Security and privacy in mobile crowdsourcing networks: challenges and opportunities," *IEEE Communications Magazine*, vol. 53, no. 8, pp. 75–81, August 2015.

[2] Y. Guo, X. Hu, B. Hu, J. Cheng, M. Zhou, and R. Y. Kwok, "Mobile cyber physical systems: current challenges and future networking applications," *IEEE Access*, pp. 1-1.

[3] K. Parshotam, "Crowd computing: a literature review and definition," in *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, pp. 121–130, the Association for Computing Machinery, East London, South Africa, October 2013.

[4] Z. Ning, X. Hu, Z. Chen et al., "A cooperative quality-aware service access system for social internet of vehicles," *IEEE Internet of Things Journal*, pp. 1-1.

[5] R. K. Ganti, F. Ye, and H. Lei, "Mobile crowdsensing: current state and future challenges," *IEEE Communications Magazine*, vol. 49, no. 11, pp. 32–39, 2011.

[6] X. Hu, T. H. S. Chu, H. C. B. Chan, and V. C. M. Leung, "Vita: a crowdsensing-oriented mobile cyber-physical system," *IEEE Transactions on Emerging Topics in Computing*, vol. 1, no. 1, pp. 148–165, 2013.

[7] X. Hu, T. H. S. Chu, V. C. M. Leung, E. C.-H. Ngai, P. Kruchten, and H. C. B. Chan, "A survey on mobile social networks: applications, platforms, system architectures, and future research directions," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 3, pp. 1557–1581, 2014.

[8] X. Hu, J. Zhao, B.-C. Seet, V. C. M. Leung, T. H. S. Chu, and H. Chan, "S-aframe: agent-based multilayer framework with context-aware semantic service for vehicular social networks," *IEEE Transactions on Emerging Topics in Computing*, vol. 3, no. 1, pp. 44–63, 2015.

[9] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay, "General decidability theorems for infinite-state systems," in *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, pp. 313–321, 1996.

[10] A. Finkel and P. Schnoebelen, "Well-structured transition systems everywhere!," *Theoretical Computer Science*, vol. 256, no. 1-2, pp. 63–92, 2001.

[11] A. Kaiser, D. Kroening, and T. Wahl, "Efficient coverability analysis by proof minimization," in *CONCUR 2012—concurrency theory*, vol. 7454 of *Lecture Notes in Comput. Sci.*, pp. 500–515, Springer, Heidelberg, 2012.

[12] G. Ciardo, "Petri nets with marking-dependent arc cardinality: properties and analysis," in *Application and theory of Petri nets 1994 (Zaragoza, 1994)*, vol. 815 of *Lecture Notes in Comput. Sci.*, pp. 179–198, Springer, Berlin, 1994.

[13] C. Dufourd, A. Finkel, and P. Schnoebelen, "Reset nets between decidability and undecidability," in *International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Comput. Sci., pp. 103–115, Springer, Berlin, Germany, 1998.

[14] J. Esparza and M. Nielsen, "Decidability Issues for Petri Nets," *BRICS Report Series*, vol. 1, no. 8, 1994.

[15] S. Schmitz and P. Schnoebelen, "The power of well-structured systems," in *CONCUR 2013—concurrency theory*, vol. 8052 of *Lecture Notes in Comput. Sci.*, pp. 5–24, Springer, Heidelberg, 2013.

[16] G. Geeraerts, J.-F. Raskin, and L. Van Begin, "Expand, enlarge, and check: new algorithms for the coverability problem of WSTS," in *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, vol. 3328 of *Lecture Notes in Comput. Sci.*, pp. 287–298, Springer, Berlin, 2004.

[17] E. A. Emerson and K. S. Namjoshi, "On model checking for non-deterministic infinite-state systems," in *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pp. 70–80, IEEE, 1998.

[18] J. Esparza, A. Finkel, and R. Mayr, "On the verification of broadcast protocols," in *Proceedings of the 14th Symposium on Logic in Computer Science*, pp. 352–359, IEEE, 1999.

[19] P. A. Abdulla, A. Bouajjani, and B. Jonsson, "On-the-fly analysis of systems with unbounded, lossy fifo channels," in *Proceedings of the International Conference on Computer Aided Verification*, pp. 305–318, Springer, 1998.

[20] J. Bingham and A. J. Hu, "Empirically Efficient Verification for a Class of Infinite-State Systems," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 3440, pp. 77–92, Springer, 2005.

[21] J. Kloos, R. Majumdar, F. Niksic, and R. Piskac, "Incremental, inductive coverability," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 8044, pp. 158–173, 2013.

[22] A. Kaiser, D. Kroening, and T. Wahl, "A widening approach to multithreaded program verification," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 36, no. 4, 2014.

[23] A. R. Bradley, "SAT-based model checking without unrolling," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 6538, pp. 70–87, 2011.

[24] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Proceedings of the Formal Methods in Computer-Aided Design FMCAD '11*, pp. 125–134, November 2011.

[25] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *Computer Aided Verification*, vol. 1254 of *Lecture Notes in Computer Science*, pp. 72–83, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[26] M. A. Colón and T. E. Uribe, "Generating finite-state abstractions of reactive systems using decision procedures," in *Proceedings of the International Conference on Computer Aided Verification*, vol. 1998, pp. 293–304, Springer.

[27] K. L. McMillan, *Symbolic Model Checking*, Springer US, Boston, Ma, USA, 1993, 25–60.

[28] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.

[29] J. Esparza, R. Ledesma-Garza, R. Majumdar, P. Meyer, and F. Niksic, "An SMT-based approach to coverability analysis," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 8559, pp. 603–619, 2014.

[30] T. Murata, "Petri nets: properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[31] W. Reisig, "Understanding Petri nets: Modeling techniques, analysis methods, case studies," *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*, pp. 1–230, 2013.

[32] K. Athanasiou, P. Liu, and T. Wahl, "Unbounded-thread program verification using thread-state equations," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 9706, pp. 516–531, 2016.

[33] G. Higman, "Ordering by Divisibility in Abstract Algebras," *Proceedings of the London Mathematical Society*, vol. 3-2, no. 1, pp. 326–336, 1952.

[34] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, vol. 3440 of *Lecture Notes in Computer Science*, pp. 570–574, Springer Verlag.

[35] L. Zhang, W. Hu, W. Qu, Y. Guo, and S. Li, "A Formal Approach to Verify Parameterized Protocols in Mobile Cyber-Physical Systems," *Mobile Information Systems*, vol. 2017, Article ID 5731678, 2017.

[36] E. D'Osualdo, J. Kochems, and C.-H. L. Ong, "Automatic verification of Erlang-style concurrency," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 7935, pp. 454–476, 2013.