# Brief Announcement: Verification of Stabilizing Programs with SMT Solvers

Jingshu Chen and Sandeep Kulkarni

Michigan State University,
3115 Engineering Building, 48824 East Lansing, US

**Abstract.** We focus on the verification of stabilizing programs using SMT solvers. SMT solvers have the potential to convert the verification problem into a satisfiability problem of a Boolean formula and utilize efficient techniques to determine whether it is satisfiable. In this work, we study the approach of utilizing techniques from bounded model checking to determine whether the given program is stabilizing.

**Key words:** Verification, Stabilization, Model checking

## 1 Introduction

One of the successful automated approaches is model checking [2]. Model checking is a technique to automatically verify whether a given model meets a given property. If the program does not meet the given property, the process of model checking typically produces a counterexample.

In this paper, we evaluate the effectiveness of SMT solvers in verifying stabilization with the use of bounded model checking. The process of using bounded model checking stabilization to verify consists of two parts, (1) verification of *closure* and (2) verification of *convergence*. Specifically, the former requires that if the program begins in a legitimate state then it remains in legitimate states. And, the latter requires that if the program starts in a state outside its set of legitimate states then it eventually reaches a legitimate state.

## 2 Approach for Verifying Stabilization with SMT Solvers

In this section, we present the approach of verifying self-stabilization properties with SMT solvers by utilizing techniques from bounded model checking.

Verification of stabilization consists of two parts: (1) verifying *closure* and (2) verifying *convergence*. In Section 2.1, we identify the formula whose satisfiability can be used to determine whether closure property is satisfied. In Section 2.2, we identify the formula whose satisfiability can be used to determine whether convergence property is satisifed.

### 2.1 Verifying Closure

Let $\mathcal{P}$ be the given program and let $\mathcal{I}$ be the legitimate state predicate to conclude that $\mathcal{P}$ is stabilizing. Let $\mathcal{T}$ be the predicate that characterizes transitions of $\mathcal{P}$.

Observe that the closure property requires that if $(s_0, s_1)$ is a transition of program $\mathcal{P}$ and state $s_0$ is a legitimate state then state $s_1$ is also a legitimate state. Thus, this can be captured by formula $\neg\Psi_l$, where

$\Psi_l \quad = \quad (\mathcal{I}(s_0) \wedge \mathcal{T}(s_0, s_1) \wedge \neg\mathcal{I}(s_1))$

**Remark.** For compactness, the formula $\Psi_l$ does not explicitly specify the program or the set of legitimate states that are inputs in deciding closure.

Based on whether $\Psi_l$ is satisfiable or not, we have two scenarios, $SC_1$ and $SC_2$:

1. $SC_1$ : if $\Psi_l$ is satisfiable then it proves that it is possible to begin in a legitimate state, execute a program transition and be in a state that is not a legitimate state. This implies that the closure property is not satisfied. Moreover, in this case, assignment to $s_0$ and $s_1$ (which in turn includes values of variables of the program in state $s_0$ and $s_1$) provides a counterexample.
2. $SC_2$ : if $\Psi_l$ is unsatisfiable then this implies that the closure property is satisfied.

### 2.2 Verifying Convergence

We verify convergence by checking that starting from an arbitrary state, the program, say $\mathcal{P}$, reaches a legitimate state (in $\mathcal{I}$) in $k$ steps, where $k$ is a given parameter used in the verification. Observe that the convergence property requires us to consider a sequence of states, $s_0, s_1, \cdots, s_k$ such that each successive transitions are program transitions. Moreover, to verify (negation of) convergence requirement, we require that $\mathcal{I}(s_k)$ should be false. Additionally, in this verification, we can utilize the closure requirement to add additional constraints requiring that $\mathcal{I}(s_j)$, $0 \leq j \leq k$, should be false. Additionally, in bounded model checking, one typically adds constraint about what the initial state should be. Thus, the formula $\Psi_v$ used for verifying convergence is as follows:

$$\begin{aligned}\Psi_v = \; & \mathcal{T}(s_0, s_1) \wedge \mathcal{T}(s_1, s_2) \wedge \cdots \wedge \mathcal{T}(s_{k-1}, s_k) \\ & \neg\mathcal{I}(s_0) \wedge \neg\mathcal{I}(s_1) \wedge \cdots \wedge \neg\mathcal{I}(s_k)\end{aligned}$$

Based on whether $\Psi_v$ is satisfiable or not, we have the following two scenarios:

1. $SC_3$ : if $\Psi_v$ is satisfiable, convergence cannot be achieved in $k$ steps. In this case, the number of steps needs to be increased. If the state space of the program is finite and $k$ equals the number of states in the program then this implies that the convergence property is not satisfied.
2. $SC_4$ : if $\Psi_v$ is unsatisfiable, then it proves that even if we begin in an arbitrary state, it is impossible for the program to be in an illegitimate state if it executes for $k$ steps. In other words, the convergence property is satisfied.

## 3 Study Case: K-State Token Ring Program

In this section, we study Dijkstra's K-state token ring program [1] for illustration purpose. The token ring program is as follows: The program consists of $N + 1$ processes, numbered from 0 to $N$. Each process $p.i$, $0 \leq i \leq N$, has one variable $x.i$. The domain of $x.i$ is $\{0, 1, \ldots, K - 1\}$. These processes are organized in a unidirectional ring.

The program consists of two types of actions. The first type is for process 0. This action is enabled when $x.0$ equals $x.N$. When $p.0$ executes its action, it increments $x.0$ by 1 in modulo K arithmetic. The second type of action is for process $p.i$, $i \neq 0$. This action is enabled when $x.i$ is not equal to $x.(i-1)$. When $p.i$ executes its action, it copies $x.(i-1)$. Thus, the actions are as follows:

$$\begin{array}{lll} K_0:: & x.0 = x.N & \longrightarrow \quad x.0 = (x.0+1) \ mod \ K; \\ K_i:: & x.i \neq x.(i-1) & \longrightarrow \quad x.i = x.(i-1); \end{array}$$

*Performance evaluation.* We evaluate the performance of the token ring program in Table 1. In particular, Table 1 illustrates the time for verifying the closure and the convergence property.

**Table 1.** Verification Time for $\Psi_v$ for Token Ring

| Number of nodes | state space | Number of steps for convergence | Execution time(s) for convergence | Execution time(s) for closure |
|---|---|---|---|---|
| 3 | $10^1$ | 4 | 0.008944 | 0.005617 |
| 4 | $10^2$ | 14 | 0.494496 | 0.005979 |
| 5 | $10^3$ | 25 | 214.0957 | 0.013349 |

## 4 Conclusion

We find that the effectiveness of SMT solvers in verification of stabilization is mixed. Specifically, compared with existing approaches [3, 4] that utilize BDD based model checkers to verify stabilization, the time for verification is larger with SMT solvers. However, BDD based tools require one to identify the order of program variables in the BDD. An incorrect ordering of variables can increase the verification time by orders of magnitude making it significantly worse than the corresponding verification time with SMT solvers. Also, the results in [3, 4] apply only for verifying finite state programs. By contrast, the results in this paper demonstrate the feasibility of verifying infinite state program.

## References

1. E.W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.
2. Orna Grumberg Edmund M. Clarke and Doron A. Peled. *Model Checking*. the MIT press, 2000.
3. Fuad Abujarad Jingshu Chen and Sandeep S. Kulkarni. Effect of fairness in model checking of self-stabilizing programs. In *Proceedings of OPODIS 2010*, 2010.
4. Tatsuhiro Tsuchiya, Shin'ichi Nagano, Rohayu Bt Paidi, and Tohru Kikuno. Symbolic model checking for self-stabilizing algorithms. *IEEE Trans. Parallel Distrib. Syst*, 12:81–95, 2001.