ASSESS: A Tool for Automated Synthesis of Distributed Self-Stabilizing Algorithms

Fathiyeh Faghih[1] and Borzoo Bonakdarpour[2]

[1] Department of Electrical and Computer Engineering
University of Tehran, Iran
Email: f.faghih@ut.ac.ir
[2] Department of Computer Science, Iowa State University, USA
Email: borzoo@iastate.edu

Abstract. A distributed *self-stabilizing* system is one that always recovers to its legitimate behavior with no external intervention, even if it is initialized in an arbitrary state. It is well known that designing and reasoning about the correctness of such protocols are highly tedious and complex tasks. We present Assess (Automated Synthesizer for SElf-Stabilizing Systems), a tool that automatically synthesizes distributed self-stabilizing algorithms from their high-level specification. Assess takes as input (1) the network topology of the distributed system, (2) the legitimate behavior of the system (either explicitly as a state predicate, or implicitly as a set of LTL formulas), and (3) a set of high-level requirements such as the timing model (asynchronous or synchronous) and stabilization type (weak, strong, and monotonic). The tool utilizes powerful SMT-solving techniques and returns a self-stabilizing protocol as a set of guarded commands that realize the input specification. Since the output is correct by construction, it will not need any proof correctness. We expect the designers and researchers in the area of self-stabilization to significantly benefit from the tool.

1 Introduction

A distributed *self-stabilizing* system is one that always recovers to its set of *legitimate states* (LS) with no external intervention, even if it is initialized in a state in $\neg LS$, or it leaves LS due to the occurrence of a transient fault. Moreover, the system remains in LS thereafter, if no faults occur. Self-stabilization has a wide range of applications in networking and distributed robotics [6, 22]. The concept was first introduced by Dijkstra in a seminal paper that presents three solutions for self-stabilizing mutual exclusion in a ring [4]. Twelve years later, Dijkstra published the proof of correctness for one of his proposed protocols [5] and states that proving the correctness of self-stabilization was surprisingly more tedious than he first expected. Indeed, designing a self-stabilizing algorithm from scratch and proving its correctness is highly complex and often subject to errors. This complexity motivates the need for developing effective tools that can automatically generate correct-by-construction self-stabilizing algorithms from high-level specifications.

In this paper, we introduce our tool Assess (Automated Synthesizer for SElf-Stabilizing Systems)[3]. The tool takes as input (1) the network topology of the distributed system in terms of a set of processes and their read/write restrictions in the shared-memory model, (2) the legitimate behavior of the system in the absence of faults, and (3) a set of high-level requirements such as the timing model (asynchronous or synchronous), stabilization type (weak [15], strong, or monotonic [26]), and whether the processes are symmetric. The legitimate behavior is specified either explicitly as a state predicate LS, or implicitly as a set of LTL formulas. The tool returns as output a finite-state self-stabilizing protocol as a set of guarded commands that realize the input specification (see Figure 1).

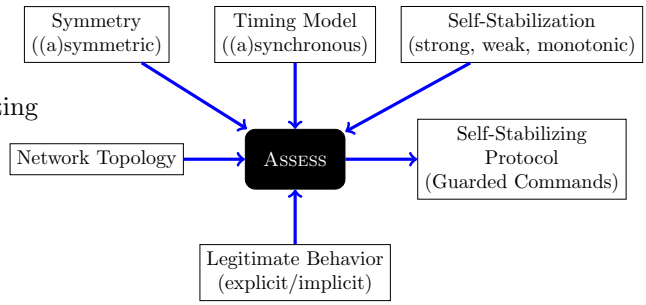

Fig. 1. Input and output of Assess.

Assess implements a collection of powerful SMT-based techniques that we proposed in [10–12]. These algorithms are inspired by bounded synthesis algorithms [14] and the tool supports the SMT-solver Z3 [1] and the model finder Alloy [18]. The internal algorithm is sound and complete, meaning that (1) a protocol synthesized by Assess is correct by construction, and (2) if the tool fails to synthesize a solution, then there does not exist one. The significance of Assess can be evaluated by its success in synthesizing a rich and well-known set of existing distributed self-stabilizing protocols. Examples include Raymond's distributed mutual exclusion algorithm [24], Dijkstra's token ring [4] (for both three and four state machines), maximal matching [21], weak stabilizing token circulation in anonymous networks [3], and the three coloring problem [16]. Therefore, we have every reason to believe that Assess will significantly assist in designing and conducting research on self-stabilizing algorithms.

ASSESS inherently has two inherent shortcomings that any such would have and they argue for more research. Scalability is a big challenge in synthesis due to its high complexity. Also, note that parameterized synthesizing a distributed self-stabilizing algorithm that works for *any* number of processes is undecidable. Currently, Assess can synthesize a small number of processes. However, it is often the case that having access to a solution for a small number of processes can give key insights to designers of self-stabilizing protocols to generalize the protocol for any number of processes. We emphasize that since our synthesis method is complete, if Assess fails to synthesize a solution, this may give the

[3] The tool can be accessed at `http://www.cas.mcmaster.ca/borzoo/assess`.

designer hints about the impossibility of finding a solution for the given problem and topology.

Organization In Section 2, we present the theoretical background of the tool. The problem of synthesis of distributed self-stabilizing protocols is discussed in Section 3. Then, Section 4 presents the high-level description of the tool. Throughout the paper, we utilize the specification of one of Dijkstra's self-stabilizing token ring protocols as a running example to demonstrate the concepts and features of our tool. Additional case studies and experimental results are presented in Section 5, and discussed in Section 6. Related work is presented in Section 7. Finally, Section 8 concludes the paper.

2 Model of Computation

2.1 Distributed Programs

Throughout the paper, let V be a finite set of discrete *variables*. Each variable $v \in V$ has a finite domain D_v. A *state* is a mapping from each variable $v \in V$ to a value in its domain D_v. We call the set of all possible states the *state space*. A *transition* in the program state space is an ordered pair (s_0, s_1), where s_0 and s_1 are two states. We denote the value of a variable v in state s by $v(s)$.

Definition 1. *A* process π *over a set V of variables is a tuple $\langle R_\pi, W_\pi, T_\pi \rangle$, where*

- *$R_\pi \subseteq V$ is the* read-set *of π; i.e., variables that π can read,*
- *$W_\pi \subseteq R_\pi$ is the* write-set *of π; i.e., variables that π can write, and*
- *T_π is the set of transitions of π, such that $(s_0, s_1) \in T_\pi$ implies that for each variable $v \in V$, if $v(s_0) \neq v(s_1)$, then $v \in W_\pi$.* □

Notice that Definition 1 requires that a process can only change the value of a variable in its write-set (third condition), but not blindly (second condition). We say that a process $\pi = \langle R_\pi, W_\pi, T_\pi \rangle$ is *enabled* in state s_0 if there exists a state s_1, such that $(s_0, s_1) \in T_\pi$.

Definition 2. *A* distributed program *is a tuple $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$, where*

- *$\Pi_\mathcal{D}$ is a set of processes over a common set V of variables, such that:*
  - *for any two distinct processes $\pi_1, \pi_2 \in \Pi_\mathcal{D}$, we have $W_{\pi_1} \cap W_{\pi_2} = \emptyset$*
  - *for each process $\pi \in \Pi_\mathcal{D}$ and each transition $(s_0, s_1) \in T_\pi$, the following* read restriction *holds:*

$$\begin{aligned} \forall s_0', s_1' : \ & ((\forall v \in R_\pi : (v(s_0) = v(s_0') \ \wedge \ v(s_1) = v(s_1'))) \wedge \\ & (\forall v \notin R_\pi : v(s_0') = v(s_1'))) \implies (s_0', s_1') \in T_\pi \end{aligned} \tag{1}$$

- *$T_\mathcal{D}$ is the set of transitions and is the union of transitions of all processes: $T_\mathcal{D} = \bigcup_{\pi \in \Pi_\mathcal{D}} T_\pi$.* □

Intuitively, the read restriction in Definition 2 imposes the constraint that for each process π, each transition in T_π depends only on reading the variables that π can read. Thus, each transition is an equivalence class in $T_\mathcal{D}$, which we call a *group* of transitions. The key consequence of read restrictions is that during synthesis, if a transition is included (respectively, excluded) in $T_\mathcal{D}$, then its corresponding group must also be included (respectively, excluded) in $T_\mathcal{D}$ as well. Also, notice that $T_\mathcal{D}$ is defined in such a way that $\mathcal{D}$ resembles an asynchronous distributed program, where process transitions execute in an *interleaving* fashion.

Example: We use the problem of *token passing* in a ring topology (*i.e.*, token ring) as a running example to describe the concepts throughout the paper. Let $V = \{x_0, x_1, x_2, x_3\}$ be the set of variables, where $D_{x_0} = D_{x_1} = D_{x_2} = D_{x_3} = \{0, 1, 2\}$. Let $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ be a distributed program, where $\Pi_\mathcal{D} = \{\pi_0, \pi_1, \pi_2, \pi_3\}$. Each process π_i $(0 \leq i \leq 3)$ can write variable x_i. Also, $R_{\pi_0} = \{x_0, x_1, x_3\}$, $R_{\pi_1} = \{x_1, x_2, x_0\}$, $R_{\pi_2} = \{x_2, x_3, x_1\}$, and $R_{\pi_3} = \{x_3, x_0, x_2\}$. Notice that following Definition 2 and read/write restrictions of π_0, (arbitrary) transitions

$$t_1 = ([x_0 = 1, x_1 = 1, x_2 = 0, x_3 = 0], [x_0 = 2, x_1 = 1, x_2 = 0, x_3 = 0])$$
$$t_2 = ([x_0 = 1, x_1 = 1, x_2 = 2, x_3 = 0], [x_0 = 2, x_1 = 1, x_2 = 2, x_3 = 0])$$

are in the same group, since π_0 cannot read x_2. This implies that if t_1 is included in the set of transitions of a distributed program, then so should be t_2. Otherwise, execution of t_1 depends on the value of x_2, which, of course, π_0 cannot read.

Definition 3. *A* computation *of $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ is an infinite sequence of states $\overline{s} = s_0 s_1 \cdots$, such that: (1) for all $i \geq 0$, we have $(s_i, s_{i+1}) \in T_\mathcal{D}$, and (2) if a computation reaches a state s_i, from where there is no state $\mathfrak{s} \neq s_i$, such that $(s_i, \mathfrak{s}) \in T_\mathcal{D}$, then the computation stutters at s_i indefinitely. Such a computation is called a* terminating computation. □

2.2 Predicates

Let $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ be a distributed program over a set V of variables. The *global state space* of $\mathcal{D}$ is the set of all possible global states of $\mathcal{D}$: $\Sigma_\mathcal{D} = \prod_{v \in V} D_v$. The *local state space* of $\pi \in \Pi_\mathcal{D}$ is the set of all possible local states of π: $\Sigma_\pi = \prod_{v \in R_\pi} D_v$.

Definition 4. *An* interpreted global predicate *of a distributed program $\mathcal{D}$ is a subset of $\Sigma_\mathcal{D}$ and an* interpreted local predicate *is a subset of Σ_π, for some $\pi \in \Pi_\mathcal{D}$.* □

Definition 5. *Let $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ be a distributed program. An* uninterpreted global predicate *up is an uninterpreted Boolean function from $\Sigma_\mathcal{D}$. An* uninterpreted local predicate *lp is an uninterpreted Boolean function from Σ_π, for some $\pi \in \Pi_\mathcal{D}$.* □

The interpretation of an uninterpreted global predicate is a Boolean function from the set of all states: $up_{\mathrm{I}} : \Sigma_D \mapsto \{true, false\}$. Similarly, the interpretation of an uninterpreted local predicate for the process π is a Boolean function: $lp_{\mathrm{I}} : \Sigma_\pi \mapsto \{true, false\}$ Throughout the paper, we use 'uninterpreted predicate' to refer to either uninterpreted global or local predicate, and use global (local) predicate to refer to interpreted global (local) predicate.

2.3 Topology

A topology specifies the communication model of a distributed program.

Definition 6. *A* topology *is a tuple* $\mathcal{T} = \langle V, |\Pi_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$*, where*

- V *is a finite set of finite-domain discrete variables,*
- $|\Pi_\mathcal{T}| \in \mathbb{N}_{\geq 1}$ *is the number of processes,*
- $R_\mathcal{T}$ *is a mapping* $\{0 \ldots |\Pi_\mathcal{T}| - 1\} \mapsto 2^V$ *from a process index to its read-set,*
- $W_\mathcal{T}$ *is a mapping* $\{0 \ldots |\Pi_\mathcal{T}| - 1\} \mapsto 2^V$ *from a process index to its write-set, such that* $W_\mathcal{T}(i) \subseteq R_\mathcal{T}(i)$*, for all* i $(0 \leq i \leq |\Pi_\mathcal{T}| - 1)$. □

Definition 7. *A distributed program* $\mathcal{D} = \langle \Pi_\mathcal{D}, T_\mathcal{D} \rangle$ has *topology* $\mathcal{T} = \langle V, |\Pi_\mathcal{T}|, R_\mathcal{T}, W_\mathcal{T} \rangle$ *iff*

- *each process* $\pi \in \Pi_\mathcal{D}$ *is defined over* V
- $|\Pi_\mathcal{D}| = |\Pi_\mathcal{T}|$
- *there is a mapping* $g : \{0 \ldots |\Pi_\mathcal{T}| - 1\} \mapsto \Pi_\mathcal{D}$ *such that*

$$\forall i \in \{0 \ldots |\Pi_\mathcal{T}| - 1\} : (R_\mathcal{T}(i) = R_{g(i)}) \ \wedge \ (W_\mathcal{T}(i) = W_{g(i)})$$

□

3 Synthesis of Distributed Self-Stabilizing Systems

We specify the behavior of a distributed self-stabilizing program based on (1) the *functional* specification, and (2) the *recovery* specification. The functional specification is intended to describe what the program is required to do in a fault-free scenario. The recovery behavior stipulates Dijkstra's idea of self-stabilization in spite of distributed control [4].

3.1 The Functional Behavior

We use linear temporal logic (LTL) [23] to specify the functional behavior of a stabilizing program. Since LTL is a commonly-known language, we refrain from presenting its syntax and semantics and continue with our running example (where $\mathbf{F}$, $\mathbf{G}$, $\mathbf{X}$, and $\mathbf{U}$ denote the 'finally', 'globally', 'next', and 'until' operators, respectively). In our framework, an LTL formula may include uninterpreted predicates. Thus, we say that a program $\mathcal{D}$ satisfies an LTL formula φ from an initial state in the set I, and write $\mathcal{D}, I \models \varphi$ iff there exists an interpretation function for each uninterpreted predicate in φ, such that all computations of $\mathcal{D}$, starting from a state in I satisfy φ. Also, the semantics of the satisfaction relation is the standard semantics of LTL over Kripke structures (i,e., computations of $\mathcal{D}$ that start from a state in I).

Example: Consider our example of *token passing* in a ring topology (*i.e.*, token ring). This problem has two functional requirements:

Safety The *safety* requirement for this problem is that in each state, only one process can execute. To formulate this requirement, we assume each process π_i is associated with a local uninterpreted predicate tk_i, which shows whether π_i is enabled. Let $LP = \{tk_i \mid 0 \leq i < n\}$. A process π_i can execute a transition, if and only if tk_i is true. The LTL formula, $\varphi_{\mathbf{TR}}$, expresses the above requirement for a ring of size n:

$$\varphi_{\mathbf{TR}} = \forall i \in \{0 \cdots n-1\} : tk_i \iff (\forall val \in \{0,1,2\} : (x_i = val) \Rightarrow \mathbf{X}\,(x_i \neq val))$$

Using the set of uninterpreted predicates, the safety requirement can be expressed by the following LTL formula:

$$\psi_{\mathbf{safety}} = \exists i \in \{0 \cdots n-1\} : (tk_i \wedge \forall j \neq i : \neg tk_j)$$

Fairness This requirement implies that for every process π_i and starting from each state, the computation should reach a state, where π_i is enabled:

$$\psi_{\mathbf{fairness}} = \forall i \in \{0 \cdots n-1\} : (\mathbf{F}\, tk_i)$$

Thus, the functional requirements of the token ring protocol is

$$\psi_{\mathbf{TR}} = \psi_{\mathbf{safety}} \wedge \psi_{\mathbf{fairness}}$$

Observe that following Definition 3, $\psi_{\mathbf{TR}}$ ensures deadlock-freedom as well.

3.2 The Problem of Synthesizing Self-Stabilizing Protocols

Definition 8. *A distributed program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ with the state space $\Sigma_{\mathcal{D}}$ is (strongly)* self-stabilizing *for an* LTL *specification ψ iff there exists a global predicate LS (called the set of* legitimate states*), such that:*

- Functional behavior: $\mathcal{D}, LS \models \psi$
- Strong convergence: $\mathcal{D}, \Sigma_{\mathcal{D}} \models \mathbf{F}\, LS$
- Closure: $\mathcal{D}, \Sigma_{\mathcal{D}} \models (LS \Rightarrow \mathbf{X}\, LS)$ □

Notice that the strong convergence property ensures that starting from any state, any computation converges to a legitimate state of $\mathcal{D}$ within a finite number of steps. The closure property ensures that execution of the program is closed in the set of legitimate states.

The problem of synthesizing a self-stabilizing algorithm is as follows. Given is (1) a topology $\mathcal{T} = \langle V, |\Pi_{\mathcal{T}}|, R_{\mathcal{T}}, W_{\mathcal{T}} \rangle$, and (2) an explicit predicate LS, or, two LTL formulas φ and ψ that involve a set LP of uninterpreted predicates. The tool is required to identify as output a distributed program $\mathcal{D} = \langle \Pi_{\mathcal{D}}, T_{\mathcal{D}} \rangle$ as a set of guarded commands for $T_{\mathcal{D}}$, such that $\mathcal{D}$ has topology $\mathcal{T}$, and (1) in the case of explicit LS, $\mathcal{D}$ is self-stabilizing for LS, or (2) in the case of implicit LS, $\mathcal{D}, \Sigma_{\mathcal{D}} \models \varphi$, and $\mathcal{D}$ is self-stabilizing for ψ. We emphasize that although we only discussed strong stabilization in Definition 8, our tool can synthesize weak [15] and monotonic-stabilizing [26] protocols as well.

4 Tool Description

In this section, we present a high-level picture of our tool: the input, internal procedure, and output (see Fig. 1).

4.1 Input to the Tool

Assess takes as input: (1) a system topology, (2) the specification of legitimate behavior, and (3) the system type; i.e., symmetry, timing model, and weak/strong/monotonic self-stabilization. It automatically generates a protocol that satisfies the given specifications. The input can be given to the tool as a plain text file or using the tool's GUI (see Fig. 2). The format of the text file can be found in the tool's user manual.

System Topology The system topology is given as a set of variable types, their finite domains, system variables (of the given types), number of processes, and read-set and write-set of each process. For example, consider a ring topology with three processes, as described in Section 2. Note that in Section 2, we considered a topology with four processes to explain read restriction, but in the sequel, to simplify the example, we consider a smaller topology with three processes. To model such a topology, we incorporate one variable per process, where each variable's domain has three values (see Fig. 2). That is, we introduce

- A variable type `t0` with three possible values: {`0,1,2`}.
- Three variables `x0`, `x1`, and `x2` of type `t0`.

The number of processes can be specified in the next row. In our example, we chose to have three processes and for convenience of reference in the paper, name them as π_0, π_1, and π_2. The user can specify the read-set (respectively, write-set) of each process by clicking on the corresponding button, and choosing the variables from the set of all defined variables (see Fig. 3). In our example, the read-set of process π_0 is {`x0,x1,x2`} (because each process on a ring has two neighbors), and its write-set is {`x0`}. The tool checks to ensure all write-sets of processes are disjoint, and also the write-set of each process is a subset of its read-set (otherwise the process will write to a variable blindly).

Note that choosing variables might be tricky in some examples. But in most cases, the user can have educated guesses about the variables and their domains from the specification (similar to programming practices). Or we can find the variables by trial and error. For example, one can start by assigning each process a boolean variable, and if no solution is found, increase the number of variables or their domains (increase the local state space of each process).

Synthesis Parameters The user can also specify the following for the output protocol:

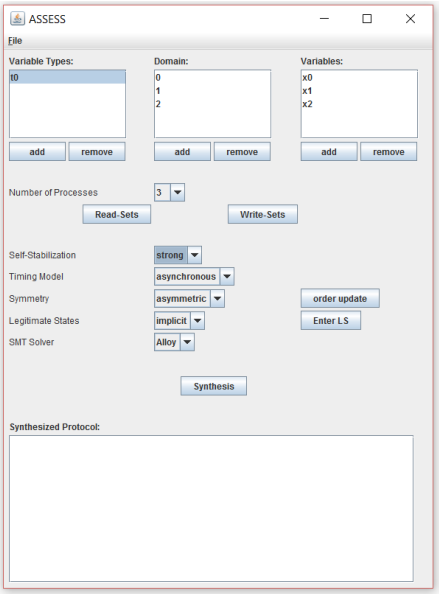

Fig. 2. ASSESS GUI

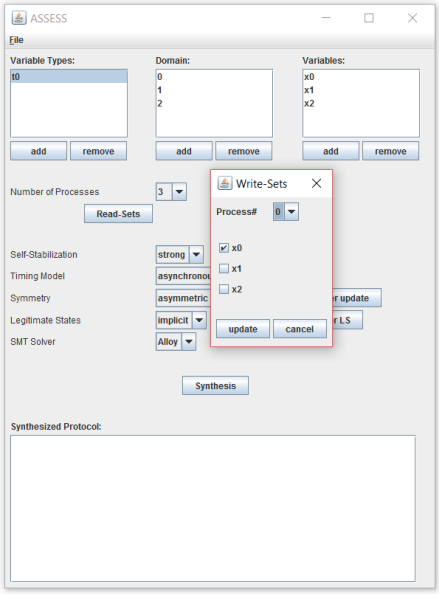

Fig. 3. Write-set specification

– *(Type of stabilization)* In a *strong-stabilizing* system, the system always recovers its legitimate behavior within a finite number of steps. A *weak-stabilizing* system [15] has only the possibility of recovery; i.e., there can be cycles along a recovery path. *Monotonic-stabilizing* [26] requires each process to change its state at most once during recovery.
– *(Timing model)* In a *synchronous* system, all *enabled* processes execute at the same time, while in an *asynchronous* system, each system transition is the execution of only one process. An asynchronous system resembles pure interleaving semantics.
– *(Symmetry)* In a *symmetric* system, all processes have the same number of elements in their read-sets and write-sets, and they all execute similarly. If the user selects to have a symmetric system, she can specify the mapping between variables of the processes' read-sets and write-sets by putting a variable ordering in the read-set and write-set of each process (see Fig. 4). This way, the variables with the same indexes are mapped to each other. For example, in our token ring example, the ordering on the read-sets can be set to the process's variable, its right neighbor's variable, and its left neighbor's variable. For example, for π_0, we have `<x0,x1,x2>`, and for π_1, we have `<x1,x2,x0>`.

Legitimate Behavior The other input to ASSESS is the legitimate behavior, which can be specified *explicitly* or *implicitly*. In the former, the user enters a Boolean predicate *LS* on the system variables to determine a subset of state space

as the set of legitimate states. In some problems, such as token ring, specification of LS as a predicate can be as difficult as finding the self-stabilizing protocol itself [12]. This motivates the idea of specifying LS implicitly. Here, the user does not need to know the exact predicate that specifies the set of legitimate states, but she can just provide a high-level specification of LS as a set of LTL formulas. As explained in Section 3, in order to keep the specification as implicit as possible, the LTL formulas can include a set of "uninterpreted predicates". These predicates are given by the user, along with a set of general constraints on them as LTL properties. For example, in the token ring problem, the user can specify an uninterpreted predicate `tki` for each process, and the process π_i changes `xi` in the next transition, if `tki` is true. As mentioned in Section 3.1, for our 3-process topology, the general constraint for `tk0` is:

```
(!tk0 && (x0==0)) => (X (x0==0))
```

Note that the constraint should be written for all three values in the domain of `x0`. Now, the implicit constraint for the legitimate behavior is:

- **Safety.** In each state, one and only one process can have the token:

  ```
  (tk0 && !tk1 && !tk2) || (!tk0 && tk1 && !tk2) ||
  (!tk0 && !tk1 && tk2)
  ```

- **Fairness.** Starting from any state, each process finally acquires the token:

  ```
  F (tk0) && F (tk1) && F (tk2)
  ```

  Another way to guarantee this requirement is that processes get enabled in a clockwise order in the ring, which can be formulated as follows:
  `(tk0 => X (tk1)) && (tk1 => X (tk2)) && (tk2 => X (tk0))`
  Note that the latter formula is a stronger constraint, and would prevent us to synthesize bidirectional protocols, such as Dijkstra's three-state solution.

The user can also specify the underlying solver to synthesize the solution. Currently, Assess supports the SMT-solver Z3 [1] and the model finder Alloy [18].

4.2 SMT-based Synthesis

Our synthesis approach is based on bounded synthesis [14] and in particular SMT-solving. Internally, Assess formulates all the required specifications given by the user as a set of SMT constraints using the techniques introduced in [10–12]. The resulting SMT instance is given to an SMT-solver to find a satisfying model. The aforementioned techniques are sound and complete. Thus, if the input instance is satisfiable, the witness model is a self-stabilizing system that realizes the input specification. If the solver returns an unsatisfiability result, we are guaranteed that there is no self-stabilizing system that can satisfy the given specification. Assess consists of four main components:

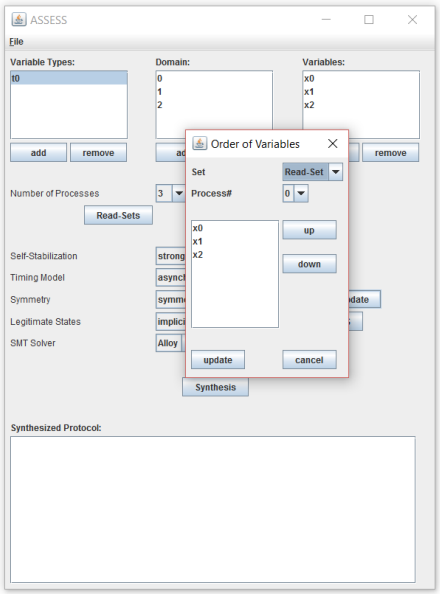

Fig. 4. Order specification

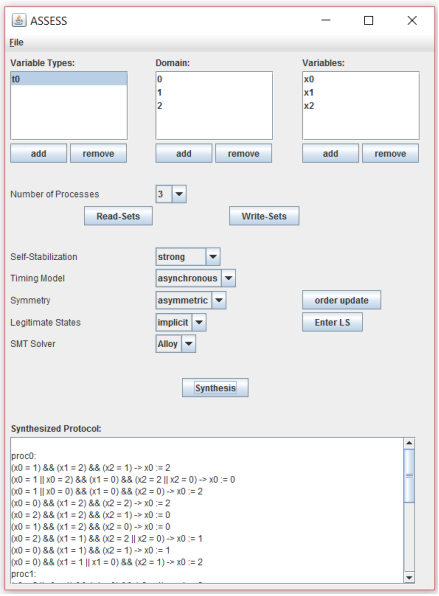

Fig. 5. Synthesis result

Input user interface We implemented ASSESS user interface using Java Graphic (see Fig. 2). The system topology, along with the desired synthesis parameters and legitimate behavior are collected by the user interface, and given to the synthesis engine.

Synthesis engine The synthesis engine generates an SMT instance based on the given input, and invokes Z3 or Alloy to solve the constraints for the generated instance. As mentioned earlier, our approach for generating the SMT instance is inspired by the concrete synthesis algorithms presented in [10–12].

SMT-solver The generated SMT instance is given to an SMT-solver to generate a witness model that represents a self-stabilizing system. ASSESS currently supports two solvers; Alloy and Z3. But it is designed in a way that other solvers can be easily plugged in the tool.

Output generator If the SMT-solver does not return "unsat", the output generator spits out the self-stabilizing protocol in terms of a set of guarded commands from the witness generated by the SMT-solver.

4.3 Output of the Tool

As mentioned earlier, ASSESS translates the synthesized transition relation by the SMT-solver into a set of guarded commands (see Fig. 5). In the case of an asynchronous system, a set of guarded commands for each process is generated, while in the case of a synchronous system, a set of guarded commands for the

whole system is generated (as all processes execute at the same time). Note that we currently do not claim that the set of output guarded commands is minimum or the recovery time is optimal.

A guarded-command is of the form `guard -> action`, where `guard` is a CNF Boolean expression, in which each clause is a disjunction of the form:

```
((vari = valj) || (vari = valz) || ...)
```

`vari` is a variable, and `valj`, `valz`, etc are values in their domains. Furthermore, `action` is a set of assignments of the form `(vari := valj)`, where `vari` is a variable, and `valj` is a value in its domain. Note that in the case of asynchronous system, for each guarded-command for the process π_i, all variables in `guard` are in the read-set of π_i, and all variables in `action` are in the write-set of π_i. As an example, the set of guarded-commands for process π_0 in the synthesized system of our token ring example is as follows:

```
proc0:

(x0 = 1) && (x1 = 2) && (x2 = 1)                       ->   x0 := 2
(x0 = 1 || x0 = 2) && (x1 = 0) && (x2 = 2 || x2 = 0) ->   x0 := 0
(x0 = 1 || x0 = 0) && (x1 = 0) && (x2 = 0)           ->   x0 := 2
(x0 = 0) && (x1 = 2) && (x2 = 2)                       ->   x0 := 2
(x0 = 2) && (x1 = 2) && (x2 = 1)                       ->   x0 := 0
(x0 = 1) && (x1 = 2) && (x2 = 0)                       ->   x0 := 0
(x0 = 2) && (x1 = 1) && (x2 = 2 || x2 = 0)           ->   x0 := 1
(x0 = 0) && (x1 = 1) && (x2 = 1)                       ->   x0 := 1
(x0 = 0) && (x1 = 1 || x1 = 0) && (x2 = 1)           ->   x0 := 2
```

5 Selected Case Studies and Experimental Results

In this section, we report the results of selected case studies that we used to evaluate our tool. The reader can find more case studies in the tool distribution. Here, we demonstrate the effectiveness of our tool by synthesizing existing well-known protocols.

5.1 Maximal Matching

Our first case study is distributed self-stabilizing *maximal matching* [17,25]. Each process maintains a *match* variable with domain of all its neighbors and its own index that indicates the process is not matched to any of its neighbors. The set of legitimate states is the disjunction of all possible maximal matchings on the given topology. As an example, for a graph of three processes connected on a line topology, we have:

```
(v0==m01 && v1==m10 && v2==m22) ||
(v0==m00 && v1==m12 && v2==m21)
```

Table 1 presents our results for different sizes of line and star topologies. Note that there is no symmetric protocol for a line or star topology.

Topology	# of Processes	Self-Stabilization	Timing Model	Solver	Time (sec)
line	3	strong	asynchronous	Alloy	0.437
line	3	strong	asynchronous	Z3	0.264
line	3	strong	synchronous	Alloy	0.273
line	3	strong	synchronous	Z3	0.119
line	4	strong	synchronous	Alloy	4.99
line	4	strong	synchronous	Z3	0.604
line	4	weak	synchronous	Alloy	4.64
line	4	weak	synchronous	Z3	0.506
star	4	strong	asynchronous	Alloy	3.98
star	4	strong	asynchronous	Z3	6.842
star	4	weak	asynchronous	Alloy	3.033
star	4	weak	asynchronous	Z3	7.962
star	5	strong	asynchronous	Alloy	89.484
star	5	strong	asynchronous	Z3	668.166

Table 1. Results for synthesizing maximal matching for line and star topologies.

5.2 The Three-Coloring Problem

In the *three-coloring problem* [16], we have a set of processes connected in a ring topology. Each process π_i has a variable $color_i$, with the domain $\{b, r, y\}$. Each value of the variable $color_i$ represents a distinct color. A process can read and write its own variable. It can also read, but not write the variables of its left and right processes. For example, in a ring of four processes, the read-set and write-set of π_0 are $R_{\mathcal{T}}(0) = \{color_0, color_1, color_3\}$ and $W_{\mathcal{T}}(0) = \{color_0\}$, respectively. The set of legitimate states is those where each process has a color different from its left and right neighbors. Thus, for a ring of four processes, LS is defined by the following predicate:

```
(color0 != color1) && (color1 != color2) &&
(color2 != color0)
```

Our synthesis results for the three coloring problem are reported in Table 2. The synthesized models for strong self-stabilization with asynchronous timing model in the symmetric case that works for 3 processes is as follows:

```
(color0 = b) && (color1 = b || color1 = y) && (color2 = b) ->  color0 := r
(color0 = b) && (color1 = b) && (color2 = r)               ->  color0 := y
(color0 = r) && (color1 = b || color1 = r) && (color2 = b) ->  color0 := b
(color0 = r) && (color1 = b || color1 = r) && (color2 = b) ->  color0 := y
```

```
(color0 = r || color0 = y) && (color1 = r) && (color2 = r) ->  color0 := b
(color0 = y) && (color1 = r || color1 = y) && (color2 = y) ->  color0 := b
(color0 = y) && (color1 = r) && (color2 = y)               ->  color0 := r
(color0 = y) && (color1 = y) && (color2 = b || color2 = r) ->  color0 := b
```

All other processes execute similarly. Additional experimental results can be found in [10–12].

# of Processes	Self-Stabilization	Timing Model	Symmetry	Solver	Time (sec)
3	weak	asynchronous	asymmetric	Alloy	2.11
3	weak	asynchronous	asymmetric	Z3	2.12
3	strong	asynchronous	symmetric	Alloy	5.95
3	strong	asynchronous	symmetric	Z3	3.245
4	weak	asynchronous	asymmetric	Alloy	51.42
4	weak	asynchronous	asymmetric	Z3	346.402

Table 2. Results for synthesizing three-coloring.

6 Discussion

Applicability: Assess requires a user to provide the tool with a set of variables and their domains as part of the network topology. Although this may seem challenging, in most cases, the user can have educated guesses about the variables and their domains from the specification (similar to programming practices), as in the maximal matching example. This process may involve some trial and error though. For example, one can start by assigning each process a Boolean variable, and if no solution is found, increase the number of variables or extend their domains.

Scalability: Obviously, scalability is an issue in our method due to high complexity of synthesis. Although our case studies deal with synthesizing a small number of processes, having access to a solution for a small number of processes may give key insights to designers of self-stabilizing protocols to generalize the protocol for any number of processes. For example, our method can be applied in cases where there exists a cut-off point [19], and we can theoretically prove that the solution works for any number of processes. Also, in cases, where we find that there is no solution for the problem, this may be a hint for a general impossibility result. Other methods can be used to improve scalability, such as counterexample-guided inductive synthesis.

7 Related Work

Related Tools FTSyn [8] is a tool for adding fault-tolerance to existing finite-state programs. It takes as input to a fault-intolerant program and a set of faults that perturbs the program. It repairs the input, so that the result is a fault-tolerant version of the input program. Sycraft (SYmboliC synthesizeR and Adder of Fault-Tolerance) [2] takes as input a distributed fault-intolerant program in terms of a set of processes, a set of fault actions and a safety specification. It transforms the input program to a

distributed fault-tolerant program. Assess is different from FTSyn and Sycraft, in that they both take as input a program, while Assess takes as input the topology, the set of legitimate states, and the program type. Also, the output of Assess is a self-stabilizing program, compared to a fault-tolerant program, which is the result of the both mentioned tools. The tool STSyn implements the heuristics presented in [13]. Unlike the algorithms implemented in Assess, STSyn algorithms are sound but not complete. Unbeast [9] is a tool for synthesis of finite-state systems from ltl formula. The tool combines the ideas in bounded synthesis, specification splitting, and symbolic game solving with binary decision diagrams (BDDs). There are similarities between bounded synthesis and our work. You can find the full comparison below.

Bounded Synthesis In bounded synthesis [14], given is a set of ltl properties. They are translated to a universal co-Büchi automaton, and then a set of SMT constraints are derived from the automaton. Our work is inspired by this idea for finding the SMT constraints for strong convergence. We were also inspired by this idea to propose a method for synthesizing weak convergence (although weak-convergence cannot be expressed in ltl). Also, our idea for synthesizing asynchronous systems is different from what is discussed in [14]; we used one transition function for each process in an asynchronous setting, while in [14], the constraints are added to the system transition function. One of the other main differences between the theoretical foundation of Assess and bounded synthesis is the ability of users to define uninterpreted predicates, and use them to specify legitimate states. It will give much more flexibility to the users, as can be seen in examples of [12]. Also, the main idea in bounded synthesis is to put a bound on the number of states in the resulting state-transition systems, and then increasing the bound if a solution is not found. In our work, since the purpose is to synthesize a self-stabilizing system, the bound is the number of all possible states, derived from the given topology. It is worth noting that one may argue that the underlying theoretical problem that Assess solves can be solved by a transformation from bounded synthesis. While this argument is valid, we emphasize that users of Assess are typically designers of self-stabilizing protocols and not experts in how to express the elements of a self-stabilizing protocol in the bounded synthesis framework.

Synthesis of Self-Stabilizing Systems In [20], the authors show that adding strong convergence is NP-complete in the size of the state space, which itself is exponential in the size of variables of the protocol. Ebnenasir and Farahat [7] also proposed an automated method to synthesize self-stabilizing algorithms. Their proposed method is not complete for strong self-stabilization. This means that if it cannot find a solution, it does not necessarily imply that there does not exist one. However, in our method, if the SMT-solver declares "unsatisfiability", it means that there is no self-stabilizing algorithm satisfying the given input constraints. Also, using our approach, one can synthesize synchronous and asynchronous programs, while the method in [7] synthesizes asynchronous systems only. Also, using our approach, the user can express legitimate states implicitly, which is not possible using the approach in [7]. Finally, our method is based on the technique of SMT solving, which is constantly evolving, and hence, we expect our technique to become more efficient as more efficient SMT solvers emerge.

8 Conclusion

We presented the tool Assess for automatic synthesis of distributed self-stabilizing systems. The tool takes as input the network topology of processes as well as the

fault-free behavior of the system, and generates as output the transition relation of the satisfying system as a set of guarded-commands, if there exists one. Our approach is to formulate the given specification as an SMT instance, and call an SMT-solver to generate a satisfying model. Assess currently supports Z3 and Alloy solvers, but other solvers can also be easily embedded. We expect our tool to significantly facilitate design and implementation of self-stabilizing algorithms. We demonstrated the effectiveness of Assess by synthesizing a set of well-known existing self-stabilizing protocols such as Raymond's distributed mutual exclusion algorithm [24], Dijkstra's token ring [4] (for both three and four state machines), maximal matching [21], weak stabilizing token circulation in anonymous networks [3], and the three coloring problem [16]. We are currently working on improving the scalability of the tool and including other types of stabilization (e.g., snap, ideal, etc). We are also attempting to use Assess to synthesize a solution for open problems in self-stabilization, where no manual solution is yet proposed.

References

1. Z3: An efficient theorem prover. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.
2. B. Bonakdarpour and S. S. Kulkarni. SYCRAFT: A tool for synthesizing fault-tolerant distributed programs. In *Concurrency Theory (CONCUR)*, pages 167–171, 2008.
3. S. Devismes, S. Tixeuil, and M. Yamashita. Weak vs. self vs. probabilistic stabilization. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 681–688, 2008.
4. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
5. E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.
6. S. Dolev and E. Schiller. Self-stabilizing group communication in directed networks. *Acta Informatica*, 40(9):609–636, 2004.
7. A. Ebnenasir and A. Farahat. A lightweight method for automated design of convergence. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 219–230, 2011.
8. A. Ebnenasir, S. S. Kulkarni, and A. Arora. FTSyn: a framework for automatic synthesis of fault-tolerance. *International Journal of Software Tools for Technology Transfer (STTT)*, 10(5):455–471, 2008.
9. R. Ehlers. Unbeast: Symbolic bounded synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 272–275, 2011.
10. F. Faghih and B. Bonakdarpour. SMT-based synthesis of distributed self-stabilizing systems. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 165–179, 2014.
11. F. Faghih and B. Bonakdarpour. SMT-based synthesis of distributed self-stabilizing systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(3):21, 2015.
12. F. Faghih, B. Bonakdarpour, S. Kulkarni, and S. Tixeuil. Specification-based synthesis of distributed self-stabilizing protocols. In *International Conference on Formal Techniques on Distributed Objects, Components and Systems (FORTE)*, pages 124–141, 2016.

13. A. Farahat and A. Ebnenasir. A lightweight method for automated design of convergence in network protocols. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(4):38, 2012.
14. B. Finkbeiner and S. Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer (STTT)*, 15(5-6):519–539, 2013.
15. M. G. Gouda. The theory of weak stabilization. In *International Workshop on Self-Stabilizing Systems*, pages 114–123, 2001.
16. M. G. Gouda and H. B. Acharya. Nash equilibria in stabilizing systems. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, pages 311–324, 2009.
17. S.-C. Hsu and S.-T. Huang. A self-stabilizing algorithm for maximal matching. *Information Processing Letters*, 43(2):77–81, 1992.
18. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press Cambridge, 2012.
19. S. Jacobs and R. Bloem. Parameterized synthesis. *Logical Methods in Computer Science (LMCS)*, 10(1), 2014.
20. A. Klinkhamer and A. Ebnenasir. On the complexity of adding convergence. In *International Conference on Fundamentals of Software Engineering*, pages 17–33, 2013.
21. F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching algorithm. *Theoretical Computer Science*, 410(14):1336–1345, 2009.
22. F. Ooshita and S. Tixeuil. On the self-stabilization of mobile oblivious robots in uniform rings. In *International Symposium on Stabilization, Safety, and Security of Distribted Systems (SSS)*, pages 49–63, 2012.
23. A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.
24. K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7:61–77, 1989.
25. G. Tel. Maximal matching stabilizes in quadratic time. *Information Processing Letters*, 49(6):271–272, 1994.
26. Y. Yamauchi and S. Tixeuil. Monotonic stabilization. In *On Principles of Distributed Systems (OPODIS)*, pages 475–490, 2010.