Computer Science Technical Report

A Hybrid Method for the Verification and Synthesis of Parameterized Weakly Stabilizing Protocols

Amer Tahat and Ali Ebnenasir

Michigan Technological University
Computer Science Technical Report
CS-TR-14-02
May 2014

Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu

A Hybrid Method for the Verification and Synthesis of Parameterized Weakly Stabilizing Protocols

Amer Tahat and Ali Ebnenasir

May 2014

Abstract

We present a hybrid method for verification and synthesis of parameterized self-stabilizing protocols where algorithmic design and mechanical verification techniques/tools are used hand-in-hand. The core idea behind the proposed method includes the automated synthesis of self-stabilizing protocols in a limited scope (i.e., fixed number of processes) and the use of theorem proving methods for the generalization of the solutions produced by the synthesizer. Specifically, we use the Prototype Verification System (PVS) to mechanically verify an algorithm for the synthesis of weakly self-stabilizing protocols. Then, we reuse the proof of correctness of the synthesis algorithm to establish the correctness of the generalized versions of synthesized protocols for an arbitrary number of processes. We demonstrate the proposed approach in the context of an agreement and a coloring protocol on the ring topology.

Contents

1 Introduction

Self-stabilization is an important property of dependable distributed systems as it guarantees *convergence* in the presence of transient faults. That is, from *any* state/configuration, a Self-Stabilizing (SS) system recovers to a set of legitimate states (a.k.a. *invariant*) in a finite number of steps. Moreover, from its invariant, the executions of an SS system satisfy its specifications and remain in the invariant; i.e., *closure*. Nonetheless, design and verification of convergence are difficult tasks [10, 19] in part due to the requirements of (i) recovery from arbitrary states; (ii) recovery under distribution constraints, where processes can read/write only the state of their neighboring processes (a.k.a. their *locality*), and (iii) the non-interference of convergence with closure. Methods for algorithmic design of convergence [2, 3, 16, 13] can generate only the protocols that are correct up to a limited number of processes and small domains for variables. Thus, it is desirable to devise methods that enable automated design of parameterized SS systems, where a *parameterized* system includes several sets of symmetric processes that have a similar code up to variable re-naming.

Numerous approaches exist for mechanical verification of self-stabilizing systems most of which focus on synthesis and verification of specific protocols. For example, Qadeer and Shankar [32] present a mechanical proof of Dijkstra's token ring protocol [10] in the Prototype Verification System (PVS) [34]. Kulkarni *et al.* [29] use PVS to mechanically prove the correctness of Dijkstra's token ring protocol in a component-based fashion. Prasetya [31] mechanically proves the correctness of a self-stabilizing routing protocol in the HOL theorem prover [18]. Tsuchiya *et al.* [35] use symbolic model checking to verify several protocols such as mutual exclusion and leader election. Kulkarni *et al.* [28, 7] mechanically prove (in PVS) the correctness of algorithms for automated addition of fault tolerance; nonetheless, such algorithms are not tuned for the design of convergence. Most existing automated techniques [5, 27, 14, 16] for the design of fault tolerance enable the synthesis of non-parametric fault-tolerant systems. For example, Kulkarni and Arora [27] present a family of algorithms for automated design of fault tolerance in non-parametric systems, but they do not explicitly address self-stabilization. Abujarad and Kulkarni [3] present a method for algorithmic design of self-stabilization in locally-correctable protocols, where the local recovery of all processes ensures the global recovery of the entire distributed system. Farahat and Ebnenasir [13, 16] present algorithms for the design of self-stabilization in non-locally correctable systems. Jacobs and Bloem [22] show that, in general, synthesis of parameterized systems from temporal logic specifications is undecidable. They also present a semi-decision procedure for the synthesis of a specific class of parameterized systems in the absence of faults.

The **contributions** of this paper are two-fold: a hybrid method (Figure 1) for the synthesis of parameterized self-stabilizing systems and a reusable PVS theory for mechanical verification of self-stabilization. The proposed method includes a synthesis step and a theorem proving step. Our previous work [13, 16] enables the synthesis step where we take a non-stabilizing protocol and generate a self-stabilizing version thereof that is correct by construction up to a certain number of processes. This paper investigates the second step where we use the theorem prover PVS to prove (or disprove) the correctness of the synthesized protocol for an arbitrary number of processes; i.e., *generalize* the synthesized protocol. The synthesis algorithms in [13, 16] incorporate weak and strong convergence in existing network protocols; i.e., adding convergence. *Weak* (respectively, *Strong*) convergence requires that from every state there exists an execution that (respectively, every execution) reaches an invariant state in finite number of steps. To enable the second step, we first mechanically prove the correctness of the Add_Weak algorithm from [16] that adds weak convergence. As a result, any protocol generated by Add_Weak will be correct by construction. Moreover, the mechanical verification of Add_Weak provides a reusable theory in PVS that enables us to verify the generalizability of small instances of different protocols generated by an implementation of Add_Weak. If the mechanical verification succeeds, then it follows that the synthesized protocol is in fact correct for an arbitrary number of processes. Otherwise, we use the feedback of PVS to determine why the synthesized protocol cannot be generalized and re-generate a protocol that addresses the concerns reported by PVS. We continue this cycle of *synthesize and generalize* until we have a parameterized protocol. Notice that the theory developed in mechanical proof of Add_Weak can also be reused for the mechanical verification of self-stabilizing protocols designed by means other than our synthesis algorithms. We demonstrate this reusability in the context of a coloring protocol (Section 7) and a binary agreement protocol (Appendix B).

Organization. Section 2 introduces basic concepts and presents their formal specifications in PVS. Then,

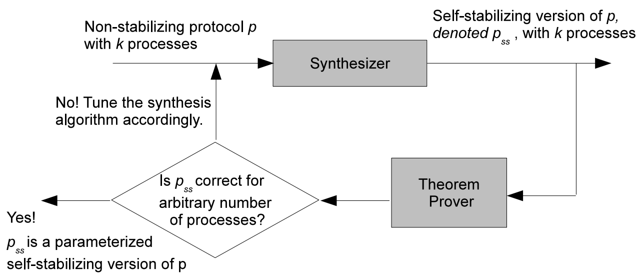
Non-stabilizing protocol *p* with *k* processes
Synthesizer
Self-stabilizing version of *p*, *denoted* p_{ss} , with *k* processes
No! Tune the synthesis algorithm accordingly.
Theorem Prover
Is p_{ss} correct for arbitrary number of processes?
Yes! p_{ss} is a parameterized self-stabilizing version of p

Figure 1: The proposed hybrid method for the synthesis of parameterized self-stabilizing protocols.

Section 3 formally presents the problem of adding convergence to protocols. Sections 4 and 5 respectively present the specification and verification of Add_Weak in PVS. Section 7 demonstrates the reusability and generalizability properties in the context of a graph coloring protocol. Section 9 discusses related work and Section 10 makes conclusing remarks and presents future extensions of this work.

2 Formal Specifications of Basic Concepts

In this section, we define protocols, state predicates, computations and convergence, and present their formal specifications in PVS. The definitions of protocols and convergence are adapted respectively from [28] and [10].

2.1 Protocols

A protocol includes a set of processes, a set of variables and a set of transitions. Since we would like the specification of a protocol to be as general as possible, we impose little constraints on the notions of state, transitions, etc. Thus, the notations *state*, *variable*, and *domain* are all abstract and nonempty. Formally, we specify them by uninterpreted types state: Type+, Variable: Type+, Dom: Type+, where '+' denotes the non-emptiness of the declared type. A *state predicate* is a set of states specified as StatePred: TYPE = set[state]. The concept of *transition* is modeled as a tuple type of a pair of states Transition: Type = [state,state] [28]. Likewise, an *action* is defined as a set of transitions, Action:Type =set[Transition]. An action can be considred as an atomic guarded command "$grd \rightarrow stmt$", where *grd* denotes a Boolean expression in terms of protocol variables and *stmt* is a set of statements that atomically update program variables when *grd* holds. We assume that Dom, Variable, and Action are finite types in our PVS specifications. A *process* is a tuple of a subset of variables that are readable by that process, a subset of variables that are writable by that process, and its subset of transitions. A *protocol prt* is a tuple of a finite set of processes, variables, and finite set of transitions.

$$\text{p\_process} : \text{TYPE+} = [\text{set[Variable]}, \text{set[Variable]}, \text{set[Transition]}]$$

$$\text{nd\_Protocol} : \text{TYPE+} = [\text{set[p\_process]}, \text{set[Variable]}, \text{set[Transition]}]$$

The *projection of a protocol prt on a state predicate I*, denoted $PrjOnS(prt, I)$, includes the set of transitions of *prt* that start in I and end in I. One can think of the projection of *prt* as a protocol that has the same set of processes and variables as those of *prt*, but its transition set is a subset of *prt*'s transitions confined in I. We model this concept by defining the following function, where Z is instantiated by transitions of *prt*. (proj_k is a built-in function in PVS that returns the k-th element of a tuple.)

$$\text{PrjOnS}(Z\text{: Action}, I\text{: StatePred})\text{: Action} =$$
$$\{t\text{:Transition} \mid t \in Z \bigwedge \text{proj\_1(t)} \in I \bigwedge \text{proj\_2(t)} \in I\}$$

2.2 Distribution and Atomicity Models

We model the impact of distribution in a shared memory model by considering read and write restrictions for processes with respect to variables. Due to inability of a process P_j in reading some variables, each transition of P_j belongs to a *group* of transitions. For example, consider two processes P_0 and P_1 each having a Boolean variable that is not readable for the other process. That is, P_0 (respectively, P_1) can read and write x_0 (respectively, x_1), but cannot read x_1 (respectively, x_0). Let $\langle x_0, x_1 \rangle$ denote a state of this program. Now, if P_0 writes x_0 in a transition $(\langle 0,0\rangle, \langle 1,0\rangle)$, then P_0 has to consider the possibility of x_1 being 1 when it updates x_0 from 0 to 1. As such, executing an action in which the value of x_0 is changed from 0 to 1 is captured by the fact that a group of two transitions $(\langle 0,0\rangle, \langle 1,0\rangle)$ and $(\langle 0,1\rangle, \langle 1,1\rangle)$ is included in P_0. In general, a transition is included in the set of transitions of a process *iff* (if and only if) its associated group of transitions is included. Formally, any two transitions (s_0, s_1) and (s'_0, s'_1) in a group of transitions formed due to the read restrictions of a process P_j meet the following constraints, where r_j denotes the set of variables P_j can read: $\forall v : v \in r_j : (v(s_0) = v(s'_0)) \wedge (v(s_1) = v(s'_1))$ and $\forall v : v \notin r_j : (v(s_0) = v(s_1)) \wedge (v(s'_0) = v(s'_1))$, where $v(s)$ denotes the value of a variable v in a state s. To enable the reusability of our PVS specifications, we specify our distribution model as a set of axioms so one can mechanically prove convergence under different distribution and atomicity models.

In the following formal specifications, v is of type Variable, p is of type p_process, t and t' are of type Transition, and non_read and transition_group are functions that respectively return the set of unreadable variables of the process p and the set of transitions that meet $\forall v : v \in r_j : (v(s_0) = v(s'_0)) \wedge (v(s_1) = v(s'_1))$ for a transition $t = (s_0, s_1)$ and its groupmate $t' = (s'_0, s'_1)$.

AXIOM subset?(proj_2(p),proj_1(p)) // *Writeable variables are a subset of readable variables.*

AXIOM member(t',transition_group(p, t, prt)) AND member(v,Non_read(p, prt))
IMPLIES Val(v,proj_1(t)) = Val(v,proj_2(t)) AND Val(v,proj_1(t')) = Val(v,proj_2(t'))

member(x,X) and subset?(X,Y) respectively represent the membership and subset predicates in a set-theoretic sense.

2.3 Computation

A *computation* of a protocol *prt* is a sequence A of states, where $(A(i), A(i+1))$ represents a transition of *prt* executed by some action of *prt*. In a more general term, a computation of any set of transitions Z, is a sequence of states in which every state can be reached from its predecessor by a transition in Z. Thus, we define the following function to return the set of computations generated by the set of transitions Z.

COMPUTATION(Z: Action): set[sequence[state]]={ A: sequence[state] $|\forall(n : nat) : ((A(n), A(n+1)) \in Z)\}$

A *computation prefix* of a protocol *prt* is a *finite* sequence of states where each state is reached from its predecessor by a transition of *prt*. Kulkarni *et al.* [28] specify a prefix as an infinite sequence in which only a finite number of states are used. By contrast, we specify a computation prefix as a finite sequence type. We believe that it is more natural and more accurate to model the concept of prefix by finite sequences. Our experience also shows that modeling computation prefixes as finite sequences simplifies formal specification and verification of reachability and convergence while saving us several definitions that were required in [28] to capture the length of the prefix. In the specification of computation prefixes, we use the predicate Condi_prefix?(A,Z) that holds when all transitions $(A(i), A(i+1))$ of a sequence A belong to a set of transitions Z. The notation A‘length denotes the length of the sequence A and A‘seq(i) returns the i-th element of sequence A.

Pos_F_S: TYPE = {c:finite_sequence[state] | c‘length > 0}

Condi_Prefix?(A:Pos_F_S,Z:Action):bool= FORALL(i: below[A‘length-1]): member((A‘seq(i), A‘seq(i+1)), Z)

PREFIX(Z: Action): set[Pos_F_S]= {A:Pos_F_S | Condi_Prefix?(A,Z) }

2.4 Closure and Convergence

A state predicate I is *closed* in a protocol *prt* iff every transition of *prt* that starts in I also terminates in I [19, 4]. The closed predicate checks whether a set of transitions Z is actually closed in a state predicate I.

```
closed?(I: StatePred, Z: Action): bool = FORALL (t:Transition | (member(t,Z)) AND member(proj_1(t), I)) :
                                        member(proj_2(t), I)
```

A protocol *prt weakly converges* to a non-empty state predicate I iff from every state s, there exists at least one computation prefix that starts in s and reaches some state in I [19, 4]. A *strongly converging* protocol guarantees that every computation from s will reach some state in I. Notice that any strongly converging protocol is also weakly converging, but the reverse is not necessarily true. A protocol *prt* is weakly (respectively, strongly) self-stabilizing to a state predicate I iff (1) I is closed in *prt*, and (2) *prt* weakly (respectively, strongly) converges to I from any state.

3 Problem Statement

The problem of adding convergence (from [16]) is a transformation problem that takes as its input a protocol *prt* and a state predicate I that is closed in *prt*. The output of Problem 3.1 is a revised version of *prt*, denoted prt_{ss}, that converges to I from any state. Starting from a state in I, prt_{ss} generates the same computations as those of *prt*; i.e., prt_{ss} behaves similar to *prt* in I.

Problem 3.1 Add Convergence

- **Input***: (1) A protocol prt; (2) A state predicate I such that I is closed in prt; and (3) A property of L_s converging, where $L_s \in \{weakly, strongly\}$.*
- **Output***: A protocol prt_{ss} such that : (1) I is unchanged; (2) the projection of prt_{ss} on I is equal to the projection of prt on I, and (3) prt_{ss} is L_s converging to I. Since I is closed in prt_{ss}, it follows that prt_{ss} is L_s self-stabilizing to I.*

Previous work [19, 16] shows that weak convergence can be added in polynomial time (in the size of the state space), whereas adding strong convergence is known to be an NP-complete problem [26]. Farahat and Ebnenasir [16, 13] present a sound and complete algorithm for the addition of weak convergence and a set of heuristics for efficient addition of strong convergence. While one of our objectives is to develop a reusable proof library (in PVS) for mechanical verification of both weak and strong convergence, the focus of this paper is mainly on enabling the mechanical verification of weak convergence for parameterized systems. Algorithm 1 provides an informal and self-explanatory representation of the Add_Weak algorithm presented in [16].

Algorithm 1 : Add_Weak

```
Input: $prt$:nd_Protocol, $I$: statePred;
Output: set[Transition]; // Set of transitions of a weakly self-stabilizing version of $prt$.
1: Let $\Delta_{prt}$ be the set of transition groups of $prt$.
2: Let $\Delta_{converge}$ be the set of transition groups that adhere to read/write restrictions of processes of $prt$, but exclude any
   transition starting in $I$;
3: $\Delta_{ws} = \Delta_{prt} \cup \Delta_{converge}$;
4: $no\_Prefix := \{s : state \mid (s \notin I) \wedge$ (there is no computation prefix using transitions of $\Delta_{ws}$ that can reach a state in $I$)$\}$
5: If ($no\_Prefix \neq \emptyset$) then weak convergence cannot be added to $prt$; return;
6: return $\Delta_{ws}$;
```

Mechanical verification of the soundness of Add_Weak ensures that any protocol synthesized by Add_Weak is correct by construction. Moreover, the lemmas and theorems developed in mechanical verification of Add_Weak provide a reusable framework for mechanical verification of different protocols that we generate using our synthesis tools [16, 25]. The verification of synthesized protocols increases our confidence in the correctness of the implementation of Add_Weak and helps us to generalize small instances of weakly converging protocols to their parameterized versions.

4 Specification of Add_Weak

This section presents the highlights of the formal specification of Add_Weak in PVS. (The complete PVS specifications are available at `http://www.cs.mtu.edu/~apklinkh/st/pvs-weak/`.) We start by specifying the basic components used in the Add_Weak algorithm, namely the transition predicates $\Delta_{prt}, \Delta_{converge}$ and Δ_{ws}, and the state predicate *no_Prefix*.

Notation. In the subsequent formal specifications, we use the identifiers Delta_prt, Delta_Converge and Delta_ws corresponding to the variables $\Delta_{prt}, \Delta_{converge}$ and Δ_{ws} in Add_Weak. The function transition_groups_proc(p,prt) returns the set of transition groups of a process p of a protocol prt.

```
Delta_Converge(prt:nd_Protocol,I:StatePred):set[set[Transition]]= {gg:set[Transition] | Exists (p:p_process |
member(p,proj_1(prt))):member(gg,transition_groups_proc(p,prt)) AND FORALL (t:Transition | member(t,gg)):NOT
member(proj_1(t),I) }
```

We find it useful to define a dependent type of all prefixes A of a set of transitions Z and we call it PREFIX_T(Z). Furthermore, we formally specify the concept of reachability as follows:

```
Reach_from?(Z:Action,A:PREFIX_T(Z),s0:state,
I: StatePred):bool = Exists (j:below[A`length]): A`seq(0)= s0 AND member(A`seq(j),I)
```

The predicate Reach_from returns true iff a state predicate I is reachable from a state $s0$ using computation prefixes of Z. To specify the set of states no_Prefix, we first specify a predicate condi_no_Prefix that determines if for a protocol prt, a state predicate I and a state s_0, no state in I can be reached from s_0 by computation prefixes of prt.

```
condi_no_Prefix?(prt:nd_Protocol,I:StatePred,s0:state):bool= FORALL (g:Action, A:PREFIX_T(g)|
member(g,Delta_ws(prt,I))AND member(A,PREFIX(g)) AND A`seq(0)= s0):NOT (Reach_from?(g,A,s0,I))
```

We then specify the state predicate no_Prefix in Add_Weak as follows:

```
no_Prefix(prt:nd_Protocol,I:StatePred):set[state]= {s0:state | NOT( member(s0,I)) AND condi_no_Prefix?(prt,I,s0)}
```

We also specify Add_Weak as a function that returns a set of transitions.

```
Add_weak(prt:nd_Protocol,
I:{II:StatePred | closed?(II,proj_3(prt))}):set[Transition] = COND empty?(no_Prefix(prt,I)) − > Delta_ws(prt,I), ELSE
− > proj_3(prt) ENDCOND
```

5 Verification of Add_Weak

In order to prove the soundness of Add_Weak, we check if (1) I is unchanged; (2) the projection of Δ_{ws} on I is equal to the projection of Δ_{prt} on I, and (3) Δ_{ws} is weakly converging to I. The first constraint holds trivially since no step of Add_Weak adds/removes a state to/from I. Next, we present a set of lemmas and theorems that prove the other two constraints of Problem 3.1.

5.1 Verifying the Equality of Projections on Invariant

In this section, we prove that Constraint 2 of Probelm 3.1 holds for the output of Add_Weak, denoted by a protocol whose set of transitions is Δ_{ws}. Our proof obligation is to show that the projection of Δ_{ws} on I is equal to the projection of Δ_{prt} on I. We decompose this into two set inclusion obligations of PrjOnS(Delta_prt,I) $\subseteq$ PrjOnS(Delta_ws,I) and PrjOnS(Delta_ws,I) $\subseteq$ PrjOnS(Delta_prt,I). Notice that, by assumption, closed?(I,Delta_prt) is true.

Lemma 5.1 PrjOnS(Delta_prt,I) *is a subset of* PrjOnS(Delta_ws,I).

Proof. The proof is straightforward since by construction we have $\Delta_{ws} = \Delta_{prt} \cup \Delta_{converge}$. $\square$

Lemma 5.2 PrjOnS(Delta_ws,I) *is a subset of* PrjOnS(Delta_prt,I).

Proof. If a transition $t = (s_0, s_1)$ is in PrjOnS(Delta_ws,I) then $s_0 \in I$. Since $\Delta_{ws} = \Delta_{prt} \cup \Delta_{converge}$, either $t \in \Delta_{prt}$ or $t \in \Delta_{converge}$. By construction, $\Delta_{converge}$ excludes any transition starting in I including t. Thus, t must be in Δ_{prt}. Since $s_0 \in I$, it follows that $t \in$ PrjOnS(Delta_prt,I). $\square$

Theorem 5.1 PrjOnS(Delta_ws,I) = PrjOnS(Delta_prt,I). *(Proof follows from Lemmas 5.1 and 5.2.)*

5.2 Verifying Weak Convergence

In this section, we prove the weak convergence property (i.e., Constraint 3 of Problem 3.1). Specifically, we show that from any state $s_0 \in \neg I$, there is a prefix A in PREFIX(Delta_ws) such that A reaches some state in I. Again, we observe that, an underlying assumption in this section is that closed?(I,Delta_prt) holds. For a protocol *prt* and a predicate I that is closed in *prt* and a state $s \notin I$, we have:

Lemma 5.3 *If* empty?(no_Prefix(prt,I)) *holds then* condi_no_Prefix?(prt,I,s) *returns false.*

Lemma 5.4 *If* condi_no_Prefix?(prt,I,s) *returns false, then there exists a sequence of states A and a set of transitions Z such that* Z $\in$ Delta_ws*,* A $\in$ PREFIX(Z)*,* A(0)=s *holds, and* Reach_from?(Z,A,s,I) *returns true.*

Lemma 5.4 implies that when Add_Weak returns, the revised version of *prt* guarantees that there exists a computation prefix to I from any state outside I; hence weak convergence. This is due to the fact that A is a prefix of Δ_{ws}.

Theorem 5.2 *If* empty?(no_Prefix(prt,I)) *holds and $s \notin I$ then there exists a sequence of states* A *that starts from s and* A $\in$ PREFIX(Delta_ws) *and* Reach_from?(Delta_ws(prt,I),A,s,I) *returns true.*

Proof. Since the proof of this theorem is abstract, for ease of presentation we refer the reader to the proof of Theorem 7.1, which is an instantiation of this proof in the context of the token ring protocol. $\square$

6 Compound Grind Proof Technique

This section presents a novel proof technique we use in proving the lemmas and theorems in our PVS theory. The contents of this section are orthogonal to the Add_Weak PVS theory and can be applied for the proof of theorems in different settings in PVS. We refer to the proposed technique as *compound grind* abbreviated *COMP_GRIND*.

6.1 COMP_GRIND Proof Technique

One reason behind the complexity of mechanical verification and deduction is that theorems and lemmas should be proved interactively. This implies that the process of formalization has an experimental nature [32]. In particular, the choice of definitions and theorems must be performed very carefully; otherwise, the mechanical verification may easily become unmanageable. For example, in PVS, the proof rule *grind* is very powerful[1]. In many cases, *grind* can complete the proof. However, in some other cases (especially when the proof of a theorem requires long sequence of composite definitions and expansions) *grind* may fail to obtain the correct instantiations or the definition expansions. Moreover, *grind* may lead to a large set of subgoals due to the fact that the proofs of subgoals were designed to be independent from each other. Next section provides examples of the two cases.

Throughout our proof of Add_Weak, we use a greedy algorithm to give a stable proof technique [9] to help us detect the problematic formalizations. Further, to make the process more automated, we insert some

[1]The grind command can do skolemization, instantiation, if-lifting, proof simplification, rewriting using lemmas as rewrite rules, definition expansion and explicit case analysis. *grind* can perform these steps repeatedly until no further simplification is possible [34]

control points over the experimental nature of the mechanical proof process. In the proof of Add_Weak, we use the proposed method to create a reasonable number of subgoals/lemmas and to decrease the complexity of the used proof rules and strategies.

In order to prove implications of the form $Q \Rightarrow Z$ using the COMP_GRIND method, we divide the proof into a sequence of steps $Q \Rightarrow a(1)$, $Q \Rightarrow a(2), \cdots$, $Q \Rightarrow a(n)$, where $a(n) = Z$. Each step is an implication with the antecedent Q and a different consequent $a(i)$, where $a(n) = Z$. Moreover, each consequent $a(i)$ should be specified in such a way that $\forall i : 1 \leq i < n : a(i) \Rightarrow a(i+1)$ holds. We call this sequence the proof sequence of $Q \Rightarrow Z$. Further, the following two conditions are met:

1. Let $A(i)$ denote $(Q \Rightarrow a(i))$. The proof of $A(i)$ should have at most c subgoals, where c is a fixed non-negative and small number. Otherwise, the formalization of $Q \Rightarrow a(i)$ is considered problematic and we try to simplify it to meet this condition and Condition 2 below. If the reformulation technique fails to decrease the number of subgoals, then we will divide $A(i)$ into more than one node, where each has at most c subgoals. If that was impossible then the initial choice of c is updated and it becomes equal to the number of subgoals of $A(i)$.

2. $A(i)$ is proved by an appropriate instantiation for $A(i-1)$ and a small number of simple rules less than or equal to c in addition to grind.

In our experience $c < 7$ has worked well. Let $R(i)$ denote the set of rules used in the proof of node $A(i)$. Then, an element of $R(i)$ can be (i) a simple primitive rule such as *grind* or *Skosimp**; (ii) a call to a previous node, or an axiom. The cardinality of $R(i)$ is less than or equal to c. There is a trade off between the length of the proof sequence and the complexity of the proofs of the nodes. In the simplest non-trivial case one can choose R to be $\{skosimp^*, grind\}$. However, we note that the efforts of formalizing a new node $A(i)$ must not cost more than the efforts of completing the proof without $A(i)$.

6.2 Recursive Nature of COMP_GRIND

COMP_GRIND has a bottom_up recursive nature. In particular, for $Q \Rightarrow Z$, let $lemma1$ denote $A(1)$. Consider the case where $A(1)$ is proved by $(skosimp^*, grind))$. Then, the proof of $lemma(2) = A(2)$ is as follows:

$$(skosimp^*(node1(inst))), R(1), grind)$$

In general, the proof of $node(n)$ which equals to $Q \Rightarrow Z$ can be represented by the following expression:

$$(skosimp^*(call(node(1, ..., n-1)(inst))), grind, R(n-1))$$

This recursive nature of COMP_GRIND illustrates how proof efforts can be reused for proving higher-level lemmas/theorems.

6.3 Complexity of COMP_GRIND

The actual cost of the proof of $A(i)$ in terms of the number of *proof rules* is at most $2 + c$ plus the cost of proving $A(i-1)$. (The constant 2 is due to using *grind* and *skosimp*.) Since PVS saves the proof of each node in a .prf file, invoking the saved proof has a unit cost. Thus, the worst case cost of proving $A(i)$ would be $2 + c + (i - 1)$. Let n denote the length of the proof sequence. As a result, the worst case cost of proving $Q \Rightarrow Z$ would be $((2 + c) \times n) + (\frac{n^2 - n}{2})$ in terms of the number of proof rules, which would be $O(n^2)$. The proof cost in terms of *number of subgoals* is at most $c \times n$; i.e., $O(n)$.

6.4 GRIND vs COMP_GRIND

The objective of COMP_GRIND is to control the behavior of *grind* towards avoiding a large number of subgoals. For example, the initial proof of the soundness of Add_Weak required the proof of 90 subgoals, which was decreased to 4 subgoals after using the COMP_GRIND technique. Even though the proofs of some nodes are non-trivial, COMP_GRIND allows us to focus only on the proofs of the hardest parts of the original proof tree. As a result, the focus of the prover shifts to the appropriate specification of each element of the proof sequence.

7 Reusability and Generalizability

In this section, we demonstrate how the lemmas and theorems proved for the soundness of Add_Weak can be reused in proving the correctness of a graph coloring protocol and in generalizing it. Due to space constraints, we omit the proof of correctness of a binary agreement protocol (which is available in Appendix B). *Reusability* enables us to instantiate the abstract concepts/types (e.g., state predicate I and actions of a protocol) for a concrete protocol and reuse the mechanical proof of Add_Weak to prove the weak concergence of that protocol. *Generalizability* determines whether a small instance of a protocol synthesized by our implementation of Add_Weak [16] can be proven to be correct for an arbitrary number of processes.
$TR(m,n)$**: Coloring on a ring of n processes with $m > 2$ colors**. We have used the Stabilization Synthesizer (STSyn) [16] tool to automatically generate the 3-coloring protocol for rings of up to 40 processes (i.e., $n < 41$). Nonetheless, due to scalability issues, STSyn cannot synthesize a self-stabilizing 3-coloring protocol for $n > 40$. In this section, we apply the proposed approach of *synthesize in small scale and generalize* to prove (or disprove) that the synthesized 3-coloring protocol is correct for rings of size greater than 40 and with more than 2 colors (i.e., $m > 2$).

The coloring protocol, denoted $TR(m,n)$, includes $n > 3$ processes located along a bidirectional ring. Each process P_j has a local variable c_j with a domain of $m > 2$ values representing m colors. Thus, the set of variables of $TR(m,n)$ is $V_{TR(m,n)} = \{c_0, c_1, ..., c_{n-1}\}$. Each process P_j can read $\{c_{j\ominus 1}, c_j, c_{j\oplus 1}\}$, and is allowed to write only c_j, where $\oplus$ and $\ominus$ denote addition and subtraction modulo n respectively. The set of legitimate states of $TR(m,n)$ includes the states where no two neighboring processes have the same color. Formally, $I_{coloring} = \forall j : 0 \leq j < n : c_j \neq c_{j\oplus 1}$. The coloring protocol has applications in several domains such as scheduling, bandwidth allocation, register allocation, etc. It is known that if $m > d$, where d is the max degree in the topology graph of the system, then the coloring problem is solvable. For this reason, we have $m > 2$ for the ring. Using STSyn [16], we have automatically generated the following action for each process P_j $(0 \leq j < 41)$:

$$A_j : (c_j = c_{j\ominus 1}) \vee (c_j = c_{j\oplus 1}) \rightarrow c_j := other(c_{j\ominus 1}, c_{j\oplus 1}) \tag{1}$$

If P_j has the same color as that of one of its neighbours, then P_j uses the function $other(c_{j\ominus 1}, c_{j\oplus 1})$ to non-deterministically set c_j to a color different from $c_{j\ominus 1}$ and $c_{j\oplus 1}$. While $TR(3,n)$ is correct by construction for $n \leq 40$, we would like to investigate whether $TR(3,n)$ is weakly stabilizing for $n > 40$. The reuse of mechanical proof of Add_Weak greatly simplifies the proof of generalization of the synthesized protocol.

7.1 PVS Specification of Coloring

This section presents the PVS specification of $TR(m,n)$. First, we instantiate the basic types in the PVS specification of Add_Weak for the coloring protocol. Then, we present the specifications of some functions that we use to simplify the verification tasks. Finally, we specify the processes and the protocol itself.
Basic types. We first define the parameterized type COLORS: below[m] to capture the colors and the size of variable domains. Then, we define a state as a finite sequence of colors of length n; i.e., STC: NONEMPTY_TYPE {s:finseq | s`length=n}. Since each variable c_j holds two pieces of information namely the process position in the ring and the color, we model their type by the tuple type ndx_varb:TYPE+=[COLORS,below[n]]. The predicate is_nbr?(K:ndx_varb,L:ndx_varb) returns true iff K and L are

two neighboring processes; i.e., mod(abs(K‘2-L‘2),n) ≤ 1, where K‘2 denotes the second element of the pair K (which is the position of K in the ring). Likewise, we define the predicate is_bad_nbr?(K:ndx_varb,L:ndx_varb) that holds iff is_nbr?(K,L) holds and K‘1 = L‘1. To capture the *locality* of process j, we define the non-empty dependent type nbr_v(K:ndx_varb):TYPE+ = {L:ndx_varb | is_nbr?(K,L) }. Likewise, we define the type bad_nbr_v(K:ndx_varb):TYPE ={L:ndx_varb | is_bad_nbr?(K,L)} to capture the set of neighbors of a process that have the same color as that process. The function nbr_colors(K:ndx_varb):set[COLORS] returns the set of colors of the immediate neighbours of a process.

Functions. In order to simplify the verification of convergence (in Section 7.2), we associate the subsequent functions with a global state. For example, we define a function ValPos(s:STC,j:below[n]):ndx_varb=(s‘seq(j),j) that returns the value and the position of process j in a global state s as a tuple of type ndx_varb. An example use of this function is Val(s:STC,L:ndx_varb)= ValPos(s,L‘2)‘1. Moreover, the predicate nbr_is_bad?(s:STC,j:below[n]):bool = nonempty?(bad_nbr_v(ValPos(s,j))) returns true iff for an arbitrary state s and a process j the set of bad neighbors of the variable ValPos(s,j) is nonempty; we refer to such a case by saying *s is corrupted at process j*. Notice that an illegitimate state can be corrupted at more than one position. We also define the predicate nbr_is_good?(s:STC,j:below[n]):bool as the negation of the predicate nbr_is_bad?(s:STC,j:below[n]):bool. Using the aforementioned functions, we also define the set of illegitimate states as S_ill:TYPE= { s:STC | not is_LEGT?(s)}, where is_LEGT?(s:STC):bool= Forall (j:below[n]): nbr_is_good?(s,j). The association of a global state to functions and types enables us to import Add_Weak_Conv theory with the following types [STC,below[m], ndx_varb, [ndx_varb, STC $\rightarrow$ below[m]]] to reuse its already defined types and functions in specifying $TR(m,n)$ as follows.

Specification of a process of TR(m,n). For an arbitrary global state s and a process j, we define the function READ_p which returns all readable variables of process j.

READ_p(s:STC, j:below[n]): set[ndx_varb]= {L:nbr_v(ValPos(s,j)) | TRUE } .

Similarly, we define the function WRITE_p which returns the variables that process j can write.

WRITE_p(s:STC,j:below[n]): set[ndx_varb]= {L:ndx_varb | L = ValPos(s,j)}

We now define the action of a process j as a function that returns the set of transitions belonging to that action.

DELTA_p(s:STC,j:below[n]):set[Transition] ={tr: Transition | Exists (c:bad_nbr_v(ValPos(s,j))): tr = (s,action(s,j,ValPos(s,j),c)) }

Intuitively, the function DELTA_p returns the set of transitions that belong to process j if process j is corrupted in the global state s; i.e., j has a bad neighbor. The function action(s,j,ValPos(s,j),c) returns the state reached when process j acts to correct its corrupted state. Formally, we define action(s,j,ValPos(s,j),c) as follows: (The LAMBDA abstractions in PVS enable us to specify binding expressions similar to quantified statements in predicate logic.)

action(s:STC,j:below[n],K:ndx_varb, C:bad_nbr_v(K)): STC = (# length := n, seq := (LAMBDA (i:below[n]):IF i= j THEN other(ValPos(s,j)) ELSE s(i) ENDIF) #)

We specify the function *other* to randomly choose a new color other than the corrupted one. To this end, we use the *epsilon* function over the full set of colors minus the set of colors of the neighbors of the corrupted process. Formally, we have other(K:ndx_varb):COLORS = epsilon(difference(fullset_colors,nbr_colors(K))), where fullset_colors:set[COLORS]= {cl:COLORS | TRUE}. We now present the specification of a process of the protocol $TR(m,n)$.

PRS_p(s:STC,j:below[n]): p_process = (READ_p(s,j), WRITE_p(s,j),DELTA_p(s,j))

The parameterized specification of the TR(m,n) protocol. We define the $TR(m,n)$ protocol as the type TR_m_n(s:STC):nd_Protocol =(PROC_prt(s),VARB_prt(s),DELTA_prt(s)), where the parameters are defined as follows:

PROC_prt(s:STC): set[p_process]={p:p_process | Exists (j:below[n]):p = PRS_p(s,j)}

VARB_prt(s:STC): set[ndx_varb]= {v:ndx_varb | Exists (j:below[n]):member(v, WRITE_p(s,j))}

Delta_prt(s:STC): set[Transition]= {tr:Transition | Exists (j:below[n]):member(tr,DELTA_p(s,j))} .

7.2 Mechanical Verification of Parameterized Coloring

We now prove the weak convergence of $TR(m,n)$ for $m > 2$ and $n > 40$. To this end, we show that the set no_Prefix of $TR(m,n)$ is actually empty. The proof of emptiness of no_Prefix is based on a prefix constructor function that demonstrates the existence of a computation prefix σ of $TR(m,n)$ from any arbitrary illegitimate state s such that σ includes a state in I. Subsequently, we instantiate Theorem 5.2 for $TR(m,n)$. Please see the Appendix A for the details of the proof.

Theorem 7.1 *Let prt be $TR(m,n)$. If* closed?(I,(prt)'3) *and* empty?(no_Prefix(prt,I)) *hold and $s \notin I$ then there exists a sequence of states A that starts from s and $A \in$* PREFIX(Delta_ws(prt,I)) *and* Reach_from?(A,s,I) *returns true.*

Proof. We show that Add_Weak will generate a weakly stabilizing version of protocol $TR(m,n)$ for any $n > 3$ and $m > 2$. To this end, we show that $TR(m,n)$ has an empty no_Prefix set and instantiate Theorem 5.2 for $TR(m,n)$.

Emptiness of no_PREFIX **for** $TR(m,n)$. To prove the weak convergence of $TR(m,n)$, we show that the set no_Prefix of $TR(m,n)$ is empty. Let s be an arbitrary illegitimate state. We define a sequence of states σ that starts in s and terminates in I such that all transitions of σ belong to proj_3(TR(m,n)). Before building a prefix from a particular illegitimate state s, we would like to identify a segment of the ring that is correct in the sense that the local predicate $((c_i \neq c_{i-1}) \wedge (c_i \neq c_{i+1}))$ is correct for processes from 1 to j; i.e., $\forall i : 1 \leq i \leq j : ((c_i \neq c_{i-1}) \wedge (c_i \neq c_{i+1}))$. For this purpose, we define three auxiliary functions. The first one is a corrector function that applies the *other* function on ValPos(s,j) if process j is corrupted; otherwise, it leaves c_j as is.

corrector(s:S_ill, j:below[n]):COLORS = COND nbr_is_good?(s,j) $\rightarrow$ s'seq(j), nbr_is_bad?(s,j) $\rightarrow$ other(ValPos(s,j)) ENDCOND

sc(s:S_ill,j:below[n]):STC= (# length := n, seq := (LAMBDA (i:below[n]):IF i $\leq$ j THEN corrector(s,i) ELSE s'seq(i) ENDIF) #)

The function sc(s,j) takes an illegitimate state s and an index j of the ring, and returns a global state where all processes from 1 to j have good neighbors. The rest of the processes have the same state as in s. Since we model a global state of a ring of n processes as a sequence of n colors, we can represent the application of the sc function on the j-th process in a global state s (denoted sc(s,j)) as $\langle$ corrector(s,0),...,corrector(s,j),s(j+1),...,s(n-1)) $\rangle$, where the colors of processes from $j+1$ to $n-1$ remain unchanged by sc.

The function AA below is especially useful because constructing the appropriate computation prefix from s to some invariant state in I directly is not straightforward. Since for all $j < n$ we do not know whether applying the corrector function on a state s:S_ill at a process j will result in a legitimate state, we use AA to build a sequence of states of length n formed by applying the corrector function consecutively at all processes regardless of being corrupted or not.

AA(s:S_ill): Pos_F_S = (# length := n, seq := (LAMBDA (i:below[n]): sc(s,i)) #)

Each element j in the sequence that AA(s) returns is the image of the function sc(s,j), which is a state that is correct up to process j. We define the function min_lgt(AA(s)) over the sequence AA(s) to return the minimum index for which the corresponding state is legitimate.

lgtStatesIndices(A:Pos_F_S):set[below[A'length]]={i:below[A'length] | is_LEGT?(A'seq(i)) }

min_Index(A:Pos_F_S):{i:integer | i$\geq$-1} = COND not empty?(lgtStatesIndices(A)) $->$ min(lgtStatesIndices(A)), else $->$ -1 ENDCOND

The Prefix-Constructor function. Now we are ready to define the constructor function of the required prefix from any s:S_ill to I as follows:

constPrefix(s:S_ill):Pos_F_S = (# length:= min_Index(AA(s))+2, seq:= (LAMBDA (i:below[min_Index(AA(s))+2]): if i=0 then s else sc(s,i-1) endif) #)

The last element of constPrefix(s) is the first legitimate state of AA(s). Thus, all states before the last state in constPrefix(s) are illegitimate. Moreover, each application of the corrector function corresponds to a transition that starts in an illegitimate state. This is true until reaching a legitimate state. Thus, all involved transitions in the sequence AA(s) are in $TR(m,n)$, thereby making constPrefix(s) a computation prefix of $TR(m,n)$. To show the correctness of this argument, it is sufficient to show the existence of at least one index that is equal to min_lgt(AA(s)). Thus, the sequence constPrefix(s) is well-defined for any state s outside I and reaches I as required. Thus, it is sufficient for weak convergence to show the correctness of the following two properties of the function sc.

1. If a process j is corrupted then the color of process j in the state sc(s,n-1) is the same as what the corrector function selects for it; i.e., ValPos(sc(s,n-1), j)‘1= other(ValPos(s,j)).

2. If a process j is corrupted then in the state sc(s,n-1) the process j is not corrupted (i.e., does not have a corrupted neighbor as well).

Notice that by the above two properties we show that the state generated by sc(s,n-1) is legitimate. We prove these properties as two lemmas.

Lemma 7.1 *If* nonempty?(bad_nbr_s(ValPos(s,j))) *holds then* ValPos(sc(s,n-1), j)‘1 = other(ValPos(s,j)).

Lemma 7.2 *If* nonempty?(bad_nbr_s(ValPos(s,j))) *holds then* empty?(bad_nbr_s((other(ValPos(s,j)), j))).

The mechanical proofs of these lemmas follow directly by expanding the required definitions and using the following axiom of choice.

AXIOM nonempty?(bad_nbr_s(ValPos(sl,j))) IMPLIES empty?(C: ndx_varb | is_bad_nbr?((other(ValPos(sl, j)), j), C))

We need the axiom of choice because we use the epsilon function in the definition of the other() function. This way we show there is always a different color that can correct the locality of a corrupted process.

Theorem 7.2 *Let prt be $TR(m,n)$. If* closed?(I,(prt)‘3) *and* empty?(no_Prefix(prt,I)) *hold and $s \notin I$ then there exists a sequence of states A that starts from s and $A \in$* PREFIX(Delta_ws(prt,I)) *and* Reach_from?(A,s,I) *returns true.*

8 Verifying the Convergence of Binary Agreement (AG)

In this section, we prove two correctness properties of any instance of the AG protocol using PVS. First, we verify that any synthesized version of AG generated by the Add_Weak algorithm is weakly stabilizing. Second, the small synthesized AG protocol in [6] is weakly stabilizing for any $n >= 3$; i.e., the protocol in [6] is generalizability.

Example. The protocol AG(n) includes $n > 2$ processes located on a unidirectional ring. Each process p_j has a variable c_j with a domain $Dom = \{0, 1\}$. Thus, AG(n) has the set of variables $V_{AG(n)} = \{c_0, c_1, ..., c_{n-1}\}$. Each process can read but not write its left neighbor; i.e., $Read_p = \{c_{j\ominus 1}, c_j\}$ while $Write_p = \{c_j\}$. Each process has an action defined by :

$$A_j : c_{j\ominus 1} < c_j \rightarrow c_j := c_{j\ominus 1} \qquad (2)$$

If the variable c_j has a value greater than its predecessor then the process p_j sets the value of c_j to $c_{j\ominus 1}$ (which is equal to 0 due to the binary domain).

A *legitimate state* of the AG(n) protocol is a state where all variables have the same value. If the condition $(c_{j\ominus 1} < c_j)$ holds for a process j, then we call it a *locally corrupted* process; otherwise, we say the protocol is *silent* at process j.

8.1 PVS Specification of AG(n)

In our PVS specification of AG(n) there are n processes, where n is a theory parameter of type *posnat* for the model. We assume that $n > 2$. We define the type $Bin : below[2]$ to capture the domain of the variables. Moreover, we formalize a global state of AG(n) as a finite binary sequence of length n; i.e., $STC : NONEMPTY\_TYPE\{s : finseq(Bin) \mid s`length = n\}$. Furthermore, since each variable c_j has two pieces of information namely the process position and its value, we model its type by a tuple $ndx\_varb : TYPE+ = [Bin, below[n]]$.

8.1.1 Specifying the Processes

We define the set of readable variables of a process p_j by the function $READ_p$.

Definition 8.1 READ_p(s:STC, j:below[n]): set[ndx_varb]= {L:(ValPos(s,j $\ominus$ i)) | i=0,1}

Similarly, we define the set of writable variables of process j by the function $WRITE_{(AG,j)}$.

Definition 8.2 WRITE_p(s:STC,j:below[n]): set[ndx_varb]= {L:ndx_varb | L = ValPos(s,j)}

Finally, we define the set of transitions of each process p_j as the set of all possible transitions generated by the action function of p_j. The action function will not be activated on a process j unless the state was locally corrupted at process j. We define the function $action(s, j)$ such that a locally corrupted state s at process j will be mapped non-deterministically to the state generated by $action(s, j)$.

Definition 8.3

$$actions(s,j) : STC = \begin{cases} 0 & s`seq(j \ominus 1) < s`seq(j) \\ s`seq(j) & otherwise \end{cases}$$

DELTA_p(s, j) captures the set of transitions of a process j originated at a global state s.

Definition 8.4 DELTA_p(s:S_ill, j:below[n]): set[Transition] = { tr: Transition | tr = (s, action(s,j)) }

For an arbitrary global state s, we now define a process of AG(n).

Definition 8.5 PRS_p(s:STC,j:below[n]): p_process = (READ_p(s,j), WRITE_p(s,j), DELTA_p(s,j))

8.1.2 Specifying the Parameterized Protocol AG(n)

We parameterize the definition of the protocol AG(n) with an arbitrary illegitimate state s. To capture the set of processes of the AG(n) protocol, we define the function $PROC\_prt$ as follows:

Definition 8.6 PROC_prt(s:STC): set[p_process]={p:p_process | Exists (j:below[n]):p = PRS_p(s,j) }

The function $VARB\_prt$ returns the set of variables of the protocol AG(n).

Definition 8.7 VARB_prt(s:S_ill): set[ndx_varb]= {v:ndx_varb | $\exists$ (j:below[n]):member(v, WRITE_p(s,j))}

Likewise, the function DELTA_prt returns the set of transitions of the protocol AG(n).

Definition 8.8 DELTA_prt(s:STC): set[Transition]= {tr:Transition | $\exists$ (j:below[n]):tr $\in$ DELTA_p(s,j)}

Thus, starting from an initial state s the formalization of AG(n) is given by defining AG(n) as a function of type nd_Protocol with the images of READ_prt, WRITE_prt and DELTA_prt functions over the state s as the components of AG(n).

```
AG_n_s:nd_Protocol= ( PROC_prt (s), VARB_prt (s), DELTA_prt(s) )
```

8.2 Weak Stabilization of Binary Agreement Protocol AG(n)

In this section, we show that the Add_Weak algorithm generates a weakly stabilizing version of the protocol AG(n) for any $n > 2$. To this end, we reuse Theorem 5.2 by showing that AG(n) has an empty no_Prefix.

Theorem 8.1 *Let prt_1 be an instance of the AG(n) protocol for some $n > 2$. If* closed?(I,(prt_1)`3) *and* empty?(no_Prefix(prt_1,I)) *hold and $s \notin I$ then there exists a sequence of states A that starts from s and $A \in$* PREFIX(Delta_ws(prt_1,I)) *and the predicate* Reach_from?(Delta_ws(prt_1,I),A,s,I) *returns true.*

This implicitly means that from any state outside the set of legitimate states I of the protocol AG(n) (for any $n > 2$), there exists a prefix that reaches I. We reuse the same construction that we used in the proof of Theorem 5.2 in Appendix A to show that the set no_Prefix of AG(n) is empty. The PVS specifications and proofs are available at `http://cs.mtu.edu/~apklinkh/st/pvs-weak/`.

9 Discussion and Related Work

This section discusses the impact of the proposed approach and the related work.

Abstract specification. Self-stabilization is an important property for networked systems, be it a network-on-chip system or the Internet. There are both hardware [11] and software systems [12] that benefit from the resilience provided by self-stabilization. Thus, it is important to have an abstract specification of self-stabilization that is independent from hardware or software. While several researchers [32, 29] have utilized theorem proving to formally specify and verify the self-stabilization of specific protocols, this paper presents a problem-independent specification of weak convergence that enables potential reuse of efforts in the verification of convergence of different protocols.

Impact of mechanical verification of Add_Weak. Proving the correctness of Add_Weak has several important impacts. First, any algorithm generated by Add_Weak is correct by construction because we have mechanically proved the soundness of Add_Weak in PVS. Second, after we synthesize small instance of weakly stabilizing protocols, we reuse the mechanical proof of Add_Weak to prove their correctness towards generalizing them. Such proofs increase our confidence in the reliability of the *implementation* of Add_Weak. Third, we will reuse the proof of Add_Weak towards formal specification and verification of a family of synthesis heuristics that we have developed (in [16]) for the addition of strong stabilization. Moreover, we are investigating a backtracking algorithm for the synthesis of strongly stabilizing protocols, which we will formally specify and verify in PVS by reusing the lemmas and theorems we have proved in this paper.

Generalization. One of the fundamental impediments before automated synthesis of self-stabilizing protocols from their non-stabilizing versions is the scalability problem. While there are techniques for parameterized synthesis [22, 24] of concurrent systems, such methods are not directly useful for the synthesis of self-stabilization due to several factors. First, such methods are mostly geared towards synthesizing concurrent systems from formal specifications in some variant of temporal logic. Second, in the existing parameterized synthesis methods the formal specifications are often parameterized in terms of local liveness properties of individual components (e.g., progress for each process), whereas convergence is a global liveness property. Third, existing methods often consider the synthesis from a set of initial states that is a proper subset of the state space rather than the entire state space itself (which is the case for self-stabilization). With this motivation, our contributions in this paper enable a hybrid method based on synthesis and theorem proving that enables the generalization of small instances of self-stabilizing protocols generated by our tools [25].

Related work. Kulkarni and Bonakdarpour's work [28, 7] is the closest to the proposed approach in this paper. As such, we would like to highlight some differences between their contributions and ours. First, in [28], the authors focus on mechanical verification of algorithms for the addition of fault tolerance to concurrent systems in a high atomicity model where each process can read and write all system variables in one atomic step. One of the fault tolerance requirements they consider is nonmasking fault-tolerance, where a nonmasking system guarantees recovery to a set of legitimate states from states reachable by faults and not necessarily from the entire state space. Moreover, in [7], Kulkarni and Bonakdarpour investigate the mechanical verification of algorithms for the addition of multiple levels of fault tolerance in the high atomicity model. In this paper, our focus is on self-stabilization in distributed systems where recovery should be provided from any state and high atomicity actions are not feasible.

Methods for the verification of parameterized systems can be classified into four major approaches. Abstraction techniques [21, 30, 15] generate a finite-state model of a parameterized system and then reduce the verification of the parameterized system to the verification of its finite model. Network invariant approaches [36, 23, 20] find a process that satisfies the property of interest and is invariant to parallel composition. Logic program transformations and inductive verification methods [33, 17] encode the verification of a parameterized system as a constraint logic program and reduce the verification of the parameterized system to the equivalence of goals in the logic program. In regular model checking [8, 1], system states are represented by grammars over strings of arbitrary length, and a protocol is represented by a transducer.

10 Conclusion and Future Work

This paper focuses on exploiting theorem proving for the generalization of synthesized self-stabilizing protocols that are correct in a finite scope (i.e., up to a small number of processes). We are particularly interested in weak stabilization where reachability to legitimate states is guaranteed from any state. The contributions of this paper comprise a component of a hybrid method for verification and synthesis of parameterized self-stabilizing network protocols (see Figure 1). Previous work [13, 16] provides algorithmic methods that take a small instance of a non-stabilizing protocol and automatically generates a self-stabilizing version thereof. This paper specifically presents a mechanical proof for the correctness of the Add_Weak algorithm from [16] that synthesizes weak convergence. This mechanical proof provides a reusable theory in PVS for the proof of weakly stabilizing systems in general (irrespective of how they have been designed). The success of mechnical proof for a small synthesized protocol shows the generality of the synthesized solution for arbitrary number of processes. We have demonstrated the proposed approach in the context of a binary agreement protocol (Appendix B) and a graph coloring procotol (Section 7).

We will extend this work by reusing the existing PVS theory for mechanical proof of algorithms (in [16]) that design strong convergence. Moreover, we are currently investigating the generalization of more complicated protocols (e.g., leader election, maximal matching, consensus) using the proposed approach.

References

[1] P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A survey of regular model checking. In *CONCUR*, pages 35–48, 2004.

[2] F. Abujarad and S. S. Kulkarni. Multicore constraint-based automated stabilization. In *11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 47–61, 2009.

[3] F. Abujarad and S. S. Kulkarni. Automated constraint-based addition of nonmasking and stabilizing fault-tolerance. *Theoretical Computer Science*, 412(33):4228–4246, 2011.

[4] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[5] P. C. Attie, anish Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):125–185, 2004.

[6] B. Bonakdarpour, A. Ebnenasir, and S. S. Kulkarni. Complexity results in revising UNITY programs. *ACM Transactions on Autonomous and Adaptive Systems*, 4(1):2009, 1–28.

[7] B. Bonakdarpour and S. S. Kulkarni. Towards reusing formal proofs for verification of fault-tolerance. In *Workshop in Automated Formal Methods*, 2006.

[8] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, pages 403–418, 2000.

[9] E. Denney, J. Power, and K. Tourlas. Hiproofs: A hierarchical notion of proof tree. *Electr. Notes Theor. Comput. Sci.*, 155:341–359, 2006.

[10] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[11] S. Dolev and Y. A. Haviv. Self-stabilizing microprocessor: analyzing and overcoming soft errors. *Computers, IEEE Transactions on*, 55(4):385–399, 2006.

[12] S. Dolev and R. Yagel. Self-stabilizing operating systems. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2. ACM, 2005.

[13] A. Ebnenasir and A. Farahat. Swarm synthesis of convergence for symmetric protocols. In *Proceedings of the Ninth European Dependable Computing Conference*, pages 13–24, 2012.

[14] A. Ebnenasir, S. S. Kulkarni, and A. Arora. FTSyn: A framework for automatic synthesis of fault-tolerance. *International Journal on Software Tools for Technology Transfer*, 10(5):455–471, 2008.

[15] E. A. Emerson and K. S. Namjoshi. On reasoning about rings. *International Journal of Foundations of Computer Science*, 14(4):527–550, 2003.

[16] 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):38:1–38:36, Dec. 2012.

[17] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *TPLP*, 13(2):175–199, 2013.

[18] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[19] M. Gouda. The theory of weak stabilization. In *5th International Workshop on Self-Stabilizing Systems*, volume 2194 of *Lecture Notes in Computer Science*, pages 114–123, 2001.

[20] O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In *Automated Reasoning*, pages 483–497. Springer, 2006.

[21] C. N. Ip and D. L. Dill. Verifying systems with replicated components in murphi. *Formal Methods in System Design*, 14(3):273–310, 1999.

[22] S. Jacobs and R. Bloem. Parameterized synthesis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 362–376, 2012.

[23] Y. Kesten, A. Pnueli, E. Shahar, and L. Zuck. Network invariants in action*. In *CONCUR 2002Concurrency Theory*, pages 101–115. Springer, 2002.

[24] A. Khalimov, S. Jacobs, and R. Bloem. Party parameterized synthesis of token rings. In *Computer Aided Verification*, pages 928–933. Springer, 2013.

[25] A. Klinkhamer and A. Ebnenasir. An embarrassingly parallel tool for automated synthesis of self-stabilization. `http://www.cs.mtu.edu/~apklinkh/protocon/index.html`.

[26] A. Klinkhamer and A. Ebnenasir. On the complexity of adding convergence. In *Proceedings of the 5th International Symposium on Fundamentals of Software Engineering (FSEN)*, pages 17–33, 2013.

[27] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 82–93, London, UK, 2000. Springer-Verlag.

[28] S. S. Kulkarni, B. Bonakdarpour, and A. Ebnenasir. Mechanical verification of automatic synthesis of fault-tolerance. *International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR), in Lecture Notes in Computer Science*, 3573:36–52, 2004.

[29] S. S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In *19th IEEE International Conference on Distributed Computing Systems - Workshop on Self-Stabilizing Systems*, pages 33–40, 1999.

[30] A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, infty)-counter abstraction. In *International Conference on Computer Aided Verification (CAV)*, pages 107–122, 2002.

[31] I. S. W. B. Prasetya. Mechanically verified self-stabilizing hierarchical algorithms. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), volume 1217 of Lecture Notes in Computer Science*, pages 399–415, 1997.

[32] S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In D. Gries and W.-P. de Roever, editors, *IFIP International Conference on Programming Concepts and Methods (PROCOMET '98)*, pages 424–443, Shelter Island, NY, June 1998. Chapman & Hall.

[33] A. Roychoudhury, K. N. Kumar, C. Ramakrishnan, I. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 172–187. Springer, 2000.

[34] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual.* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. A new edition for PVS Version 2 is released in 1998.

[35] T. Tsuchiya, S. Nagano, R. B. Paidi, and T. Kikuno. Symbolic model checking for self-stabilizing algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 12(1):81–95, 2001.

[36] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *International Workshop on Automatic Verification Methods for Finite State Systems*, pages 68–80, 1989.