State Space Abstraction for Parameterized Self-Stabilizing Embedded Systems*

Nikolaos Liveris Hai Zhou Robert P. Dick Prithviraj Banerjee
Northwestern University, Evanston IL 60208
nikos@ece.northwestern.edu
{haizhou, dickrp}@northwestern.edu
HP Labs, Palo Alto CA 94304
prith.banerjee@hp.com

ABSTRACT

Self-stabilizing systems are systems that automatically recover from any transient fault. Proving the correctness of a parameterized self-stabilizing system, i.e., a system composed of an arbitrary number of processes, is a challenging task. For the verification of parameterized systems the method of control abstraction has been developed. However, control abstraction can only be applied to systems in which each process has a fixed number of observable variables. In this article, we propose a technique to abstract a parameterized self-stabilizing system, whose processes have a parameterized number of observable variables, to a system with fixed number of observable variables. This enables the use of control abstraction for verification. The proposed technique targets low-atomicity, shared-memory, asynchronous systems. We establish the completeness of the method under reasonable conditions and demonstrate its effectiveness by applying it on a number of self-stabilizing distributed systems.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Distributed Systems*;
F.3.1 [**Theory of Computation**]: Logics and Meanings of Programs—*Specifying and Verifying and Reasoning about Programs*

General Terms

Reliability, Verification

Keywords

Abstraction, parameterized systems, self-stabilizing systems, verification, network invariants

*Supported by NSF under CNS-0613967 and CNS-0347941.

EMSOFT'08, October 19–24, 2008, Atlanta, Georgia, USA.

1. INTRODUCTION

Distributed embedded systems are widely used in many applications, e.g., control systems in cars, airplanes, or houses. There are cases in which reliability is the most important requirement of those systems. One way to guarantee that these systems tolerate transient faults is by making them self-stabilizing systems, which automatically recover from any transient fault [7]. This type of fault-tolerance is desirable in many distributed embedded systems [3, 15, 21]. Verifying the correctness of those systems is a challenging task while testing them is intractable. The main reason is that distributed self-stabilizing systems are designed to work for an arbitrary number of components, i.e., the number of components is given as a parameter to the specification of the system. One way to enable the usage of automated verification techniques, like model checking, on parameterized systems is by using abstraction. However, manual abstraction requires deep knowledge of the embedded application, formal specification, and model checking. Few designers have the required knowledge of both the application and model checking. Therefore, we propose a general abstraction technique for enabling automatic verification tools to check the correctness of self-stabilizing parameterized systems.

We distinguish two types of self-stabilizing systems; "strict-stabilizing" and "pseudo-stabilizing" systems [5]. After the last fault, a strict-stabilizing system will eventually enter a state in which it satisfies and will maintain the correctness property. Formally, let ϕ denote the correctness property, a strict-stabilizing system satisfies the LTL (Linear Temporal Logic) property $\diamond\phi$ and $\square(\phi \rightarrow \square\phi)$, starting from any fault state. A pseudo-stabilizing system will eventually get into states where the correctness property will never be violated. Pseudo-stabilization does not require that the system cannot enter a faulty state after reaching a correct state. However, eventually the system will be in correct states only. The LTL property for pseudo-stabilizing systems is given as $\diamond\square\phi$. Although pseudo-stabilization is weaker, it is sufficient for many practical applications.

The way system designers reason about the correctness of self-stabilizing systems is by considering all states in the state space to be initial states. The assumption is that the initial state is the first state after a transient fault. Therefore, the system can start from any state and must eventually recover, if no more faults occur. Convergence stairs is a common proof method for self-stabilizing systems [7,

11]. A finite sequence of predicates $p_0, ..., p_m$ is defined with $p_0 =$ True and $p_m = \phi$ being the correctness property of the system. The designer proves that if eventually always p_i is satisfied, then eventually always p_{i+1} is satisfied for all $i \in 0..m-1$. In LTL, it is $\diamond\Box p_i \rightarrow \diamond\Box p_{i+1}$. By this method, the pseudo-stabilization $\diamond\Box p_m$ (or persistence property) can be proved, and by showing that the safety property $\Box(p_m \rightarrow \Box p_m)$ also holds, strict-stabilization can also be established. Proving the liveness properties $\diamond\Box p_i \rightarrow \diamond\Box p_{i+1}$ is the hardest step in this method and, therefore, we focus on this type of properties in this paper. Most self-stabilizing systems are complicated and proving their correctness manually is not an easy task. Therefore, there is a need for automatic verification techniques for these systems.

There are two kinds of methods that are used for the verification of asynchronous systems: deductive verification and model checking. Deductive verification is an interactive verification method; the user is required to provide properties to facilitate the proof by the tool. Model checking is an automatic method. However, it is efficient only when applied to relatively small finite state systems. Therefore, abstraction is required to transform infinite or large state systems to smaller finite state systems for model checkers [12].

A parameterized system is built by parallel composition of N processes, where N can be any natural number. In many cases it is necessary to prove the correctness of a parameterized system independent of the number of processes. Since the number of those systems is infinite, abstraction techniques are needed. The parameterized system is first abstracted to a finite state system such that any predicate that holds for the finite state system will hold for the original system with any number of processes. Then model checking can be used to check the finite state system.

A number of abstraction techniques for parameterized systems has been developed for high-atomicity systems [6, 20, 4, 9]. High-atomicity systems are those in which a process can read the values of many neighbors in one atomic step. Since the implementation of such an atomic step is expensive, we target low-atomicity distributed systems. Moreover, since the correctness property is a liveness property, we focus on abstraction techniques for the verification of liveness properties on low-atomicity systems.

There are two abstraction techniques for proving liveness properties of low-atomicity parameterized systems: control abstraction [16, 12] and invisible invariants [10]. The idea behind control abstraction is to abstract away an arbitrary number of symmetric processes by using a network invariant I. Then the correctness property can be proved for the abstract system, which is composed of a finite number of processes and the network invariant [12]. A difficulty with this approach is that the network invariant must have the same set of observable variables as the system of symmetric processes abstracted by it. Therefore, if each of the N abstracted processes has one observable variable, the network invariant must have N observable variables. This difficulty has restricted the applicability of control abstraction. Control abstraction has been successfully applied on ring topologies of processes [14], in which every process has only two neighbors and, therefore, the number of input/output variables for each process is independent of N. It has also been successfully applied to systems for which the number of shared variables does not increase with the number of processes. An example is a mutual exclusion algorithm, in which all processes share only one semaphore [12].

The other approach is the method of invisible invariants [10]. The method can be used to bound the number of processes needed to prove a correctness property for a class of parameterized systems. The approach can be used for the verification of safety properties and response liveness properties, which are of the form $\Box(p \rightarrow \diamond r)$, i.e., for every state satisfying assertion p there is a future state satisfying assertion r. It is not known how other liveness properties can be checked using this method. Moreover, the method imposes a number of restrictions on the structure of the next state relation and the initial condition of the system. In some cases the number of required processes is large (e.g. 128 for the dining philosophers problem).

In this paper we present an abstraction technique that builds on the theory of control abstraction. We target systems in which the number of observable variables is a parameter. Such systems are very common in practice. One example is a networking system connected in a graph of arbitrary topology, where the number of neighbors of a process is a parameter. The system with a parameterized number of observable variables is abstracted to a system with a fixed number of observable variables, making it amenable to control abstraction.The proposed abstraction technique can be applied to non-self-stabilizing systems as well.

There is no other approach for applying the network invariant method to distributed systems with N observable variables in each process. The proposed technique enables the application of this method to self-stabilizing systems. Moreover, case studies demonstrate that our abstraction technique can be applied to distributed systems to which no other abstraction technique has been successfully applied. Furthermore, sufficient conditions under which the abstraction technique is complete are provided.

In addition to handling low-atomicity constraints, the proposed abstraction technique does not generate an abstract system whose size is exponential in the number of local states of each process, as it is the case in other works [20]. It also handles the weak fairness constraints for the abstracted processes, in contrast with existing abstraction techniques [4]. Finally, because it uses syntax manipulation, the complexity of building the transition relation is low compared to approaches that use decision procedures (MONA) [20, 4].

In Section 2 we describe our notation and the systems we consider. Section 3 gives an overview of a 3-step abstraction methodology and shows how the proposed abstraction technique can be used as part of the methodology. Section 4 explains the technique in detail. We demonstrate the effectiveness of the technique on a number of case studies in Section 5. For some theorems we include only proof sketches due to space limitations.

2. SYSTEMS AND NOTATIONS

We deal with the verification of closed parameterized systems. A closed parameterized system can be defined as

$$\mathcal{Q}(N) = \left(P\left(1, N\right) \| P(2, N) \| ... \| P\left(N, N\right)\right)_R \qquad (1)$$

In the above formula $P(1,N), P(2,N), ..., P(N,N)$ are identical processes up to renaming. The first argument denotes the id[1] and the second the number of observable variables of each process. The operator $\|$ denotes parallel asynchronous composition and $()_R$ restriction. Parallel asynchronous composition is equivalent to the interleaving semantics. This composition allows only one process to execute one atomic action in each step [12]. A restricted system is a closed system. There is no interaction between the system and its environment [12]. We are interested in proving the correctness of a parameterized systems described by (1).

In this paper we follow a similar notation to that used by Abadi and Lamport [1] for describing systems. Each system, which can be composed of one or more processes, is represented by its specification $\mathcal{Q}(N) = \langle \Sigma, F, \mathcal{N}, L \rangle$. Σ is the state space of the specification, $F \subseteq \Sigma$ is the set of initial states, $\mathcal{N} \subseteq \Sigma \times \Sigma$ is the next state relation, and L is the liveness property defined over Σ. Property L can be evaluated only on infinite sequences of states.

The state space Σ is defined by the domains of the variables in the system. The set V of the system variables is given by $V_g \cup \bigcup_{i \in 1..N} V_{L(i)} \cup \{\mathtt{sv}[i] \mid i \in 1..N\}$. The set V_g of global variables is a fixed set of variables observable to all processes. Set $V_{L(i)}$ are the local variables of process i, which only process i can read or modify. Finally, $\mathtt{sv}[i]$ is the shared variable of process i that any process in the system can read but only process i can modify.

The next state relation is defined using a set of atomic actions A. Each action α is described as the conjunction of its precondition and its effect $\alpha = \text{prec}(\alpha) \wedge \text{eff}(\alpha)$. The precondition (or enable condition) $\text{prec}(\alpha)$ is a proposition over the system variables. The effect part $\text{eff}(\alpha)$ describes the values of all system variables in the next state s', as a function of the current state s. More specifically, it can be considered the conjunct of $\epsilon(\alpha) \wedge \text{unch}(\alpha)$. In the last formula $\epsilon(\alpha)$ is a boolean combination of predicates of the form $m' = g(s)$, and $\text{unch}(\alpha)$ is the conjunction of predicates of the form $m' = m$. A predicate $m' = m$ of $\text{unch}(\alpha)$ specifies that variable m never changes value when α is executed. We say that an action "reads" a variable n, when n appears in expression $g(s)$ of a predicate $m' = g(s)$ of $\epsilon(\alpha)$. An action "modifies" or "writes" a variable m, when there is a predicate $m' = g(s)$ in $\epsilon(\alpha)$ and $g(s) \neq m$. This classification is based on the syntax and can be performed by static analysis.

A state pair $\langle s, s' \rangle \in \mathcal{N}$, if and only if there exists $\alpha \in A$, such that $\text{prec}(\alpha)$ is true for s and the pair of states $\langle s, s' \rangle$ satisfies $\text{eff}(\alpha)$. We assume there is a stuttering step $\tau \in A$ and for all states $s \in \Sigma$, $\langle s, s \rangle$ belongs to $\mathcal{N}$.

The liveness property L is a restriction imposed on the infinite behaviors of the system. It may include the conjunction of strong and weak fairness properties specified on some of the actions in A. We use $\mathcal{W}$ and $\mathcal{S}$ to represent the sets of actions with weak and strong fairness properties respectively. Then $L \triangleq \bigwedge_{\alpha \in \mathcal{W}} wf(\alpha) \wedge \bigwedge_{\alpha \in \mathcal{S}} sf(\alpha)$. The weak and strong fairness properties are defined as

$$wf(\alpha) \triangleq (\Box\Diamond\neg\text{prec}(\alpha)) \vee (\Box\Diamond(\langle\text{eff}(\alpha)\rangle))$$
$$sf(\alpha) \triangleq (\Diamond\Box\neg\text{prec}(\alpha)) \vee (\Box\Diamond(\langle\text{eff}(\alpha)\rangle))$$

The expression $\langle \text{eff}(\alpha) \rangle$ evaluates to true when action α is executed and the system's state changes [17]. Therefore, for a pair of states $\langle s, s' \rangle$, it holds $\langle s, s' \rangle \models \langle \text{eff}(\alpha) \rangle \Leftrightarrow \langle s, s' \rangle \models \text{eff}(\alpha) \wedge s' \neq s$.

A sequence $\sigma = s_0, s_1, ...$ of states, with $\sigma \in \Sigma^\omega$, is a behavior (or computation) of $\mathcal{Q}(N)$ if σ satisfies the specification $\mathcal{Q}(N) = \langle \Sigma, F, \mathcal{N}, L \rangle$. More specifically, it must hold that $s_0 \in F$, $\forall i \geq 0 : \langle s_i, s_{i+1} \rangle \in \mathcal{N}$, and $\sigma \models L$.

We use this operator $\models$ to denote that a predicate is valid for a state or a set of states. We extend its usage to temporal properties and sequences or sets of sequences. When a specification is used at the LHS and a temporal formula at the RHS, the temporal formula is valid for all behaviors of the specification.

Each variable of the set $\{\mathtt{sv}[j] | j \in 1..N\}$ is called a "shared variable". We use FS to denote the domain of each shared variable, which is a finite set. The expression $[1..N \rightarrow \text{FS}]$ represents the domain of all shared variables. We denote the set of all variables other than the shared variables as V_{nsv}, i.e., $V_{nsv} \triangleq V_g \cup \bigcup_{i \in 1..N} V_{L(i)}$. Besides the set of shared variables, the system can have a finite set V_g of variables that are observable to all processes. The cardinality of this set must be independent of N. Therefore, the variables in V_g are not preventing the application of the network invariant method and do not need to be abstracted. We consider these variables as elements of V_{nsv} and we restrict the usage of the term "shared variable" only for an element of the set $\{\mathtt{sv}[j] | j \in 1..N\}$.

If one action α can be obtained from another action β by replacing any appearance of one shared variable $\mathtt{sv}[k]$ with another shared variable $\mathtt{sv}[j]$, then α and β are called syntactically equivalent. A formal description of this relation is given in Section 4.2.

We now present the assumptions on the systems we consider and then elaborate on the reasons for making these assumptions and their implications.

- Λ1. Actions can either read or write at most one shared variable in each atomic step.

- Λ2. Each shared variable is a single-writer multi-reader variable. More specifically,

  $\forall j \in 1..N : \mathtt{sv}[j]$ can be written only by process j

- Λ3. The preconditions of the actions do not depend on the values of the shared variables. Therefore, reading or writing a shared variable can only be done by the effect part of an action.

- Λ4. There is no pair of actions that are not syntactically equivalent and have the same effect at same state. More specifically, if $\alpha \in \mathcal{W} \cup \mathcal{S}$, then for any action β with $\beta \neq \alpha$:

  $$\forall \langle s, s' \rangle \in \mathcal{N} : \langle s, s' \rangle \not\models (\langle \text{eff}(\alpha) \rangle \wedge \langle \text{eff}(\beta) \rangle)$$

[1]The ids of the processes are used only for naming convenience. No comparison is allowed between the ids. If a system uses relations on ids, we can abstract the result of the relation during the preprocessing step (Section 3).

Symbol	Definition
$cf(\alpha)$	constant fairness condition of action α
$\text{eff}(\alpha)$	the effect of an action α; it defines the next state values of the system variables
F	set of initial states of a system
FS	the finite set which is the domain of each shared variable, i.e., $\forall i \in 1..N : \texttt{sv}[i] \in \text{FS}$
H	a state predicate expressed over the variables in $V_g \cup \bigcup_{i \in 1..N} V_{L(i)} \cup \{\texttt{sv}[i] \mid i \in 1..N\}$; the correctness property we target is of the form $\Diamond\Box H \rightarrow \Diamond\Box J$
I	the network invariant generated during control abstraction
J	a state predicate expressed over the variables in $V_g \cup V_{L(1)} \cup \{\texttt{sv}[1]\}$; the correctness property we target is of the form $\Diamond\Box H \rightarrow \Diamond\Box J$
L	liveness condition of a system; evaluated only on infinite sequences
N	number of processes in the system
$\mathcal{N}$	next state relation of a system; $\langle s_1, s_2 \rangle \in \mathcal{N} \Leftrightarrow (\exists a \in A : \langle s_1, s_2 \rangle \models a)$, where A is the set of actions
$P(j, N)$	a process with id = j and N observable shared variables
$\text{prec}(\alpha)$	the precondition or enable condition of an action α
$\mathcal{Q}(N)$	parameterized system which is the input to our abstraction technique
$\mathcal{Q}_a(N)$	the system obtained after applying the abstraction technique; it has N process and two observable shared variables
$\mathcal{S}$	set of actions with strong fairness conditions
$sf(\alpha)$	strong fairness condition of action α
$\texttt{sv}[j]$	one shared variable of the system; only process j can modify this variable but all processes can read it
V_g	the set of global variables that any process can read and modify. This set has a fixed size.
$V_{L(i)}$	the set of local variables of process i, which all processes other than i cannot read or modify
V_{nsv}	the set of all variables of the system excluding the shared variables, i.e, $V_{nsv} \triangleq V_g \cup \bigcup_{i \in 1..N} V_{L(i)}$
$V_{\texttt{sv}}$	$\triangleq \{\texttt{sv}[j] \mid j \in 2..N\}$ the shared variables which are not modified by process 1; all the variables in this set are not present in the abstract system
$\mathcal{W}$	set of actions with weak fairness conditions
$wf(\alpha)$	weak fairness condition of action α
$\Pi_V(s)$	projection of the state s on the set of variables (or variable) V
Σ	state space of a system
φ	the correctness property after the application of the preprocessing step; for self-stabilizing systems it is equal to $\Diamond\Box J$

Table 1: Definition of Commonly Used Symbols

We believe that the above constraints are common among many applications. Restriction $\Lambda 1$ specifies the low-atomicity constraint. Restriction $\Lambda 2$ specifies ownership of the variables by the processes. Restriction $\Lambda 3$ has been used in other works ([18], Chapter 9). This restriction is based on the fact that reading a non-local variable is a more expensive operation than reading a local variable and, therefore, should be an atomic action. The decision of a process to execute an action should be based on local variables only. Consequently, shared variables should be copied to local variables before their value is used in the precondition of an action. Note that process j can maintain a local copy of $\texttt{sv}[j]$ and because of restriction $\Lambda 2$ the copy can be always equal to the value of the shared variable. The intuition behind $\Lambda 4$ is that any transition other than the stuttering step can be caused by only one action. However, syntactically equivalent actions are not restricted by $\Lambda 4$. Most systems with a program counter for each process satisfy the $\Lambda 4$ restriction. More specifically, if each instruction has a different successor, the effect of each action of one process is distinct. Since the program counter is a local variable of each process, the effect of each action cannot be simulated by an action of a different process. The restrictions $\Lambda 1$-$\Lambda 4$ do not need to hold for the fixed set of global variables in V_g. Therefore, we can have a fixed finite set of multi-writer variables.

We assume that the correctness property is given in the form $\Diamond\Box H \rightarrow \Diamond\Box J$. This type of condition is very common as a subgoal for self-stabilizing systems. For these systems it usually states that once the environment of a process satisfies a specific persistence condition ($\Diamond\Box H$), the process must satisfy a persistence condition ($\Diamond\Box J$). We consider process 1 to be the special process that must satisfy $\Diamond\Box J$. Therefore, J is expressed over the variables in $V_g \cup V_{L(1)} \cup \{\texttt{sv}[1]\}$.

3. OVERVIEW OF OUR APPROACH

In this section we provide a general methodology for proving the correctness of parameterized self-stabilizing systems. It consists of three main steps.

1. *Preprocessing*: Transform the system to a closed system of N processes, in which the domains of all variables are finite. Simplify the correctness property from $\Diamond\Box H \rightarrow \Diamond\Box J$ to $\Diamond\Box J$, by transforming the system to a system that satisfies $\Box H$.

2. *Enabling control abstraction*: Reduce the observable state space to a finite number of variables (Section 4).

3. *Applying control abstraction*: Develop process I that can be used as a network invariant. Use model-checking to verify that I is a correct network invariant for the system.

During the preprocessing step, data abstraction [13] can be used to reduce the domains of the variables to finite domains. Moreover, we proved that the correctness property of a self-stabilizing system can be simplified to $\Diamond\Box J$ by transforming the parameterized system to a system that always satisfies H. This is because $\Diamond\Box H \rightarrow \Diamond\Box J$ is equivalent to $\Box H \rightarrow \Diamond\Box J$ for systems whose set of initial states is a superset of the states that satisfy H. Self-stabilizing systems satisfy this condition, as their set of initial states is the set of reachable states (Section 1). Moreover, we can transform the system to always satisfy H by restricting its set of initial states to the set of states that satisfy H and its next state relation $\mathcal{N}$ to $\breve{\mathcal{N}} \triangleq \{\langle s, s'\rangle \in \mathcal{N} \mid s' \models H\}$. The correctness property of the new system is the persistence property $\Diamond\Box J$, which is expressed over the variables in $V_g \cup V_{L(1)} \cup \{\mathtt{sv}[1]\}$ (Section 2). Moreover, the system maintains the property that all reachable states are initial states.

During the control abstraction step, the network invariant I needs to satisfy the following properties

$$P(id, 2) \quad \sqsubseteq_M \quad I \tag{2}$$

$$(I \| I) \quad \sqsubseteq_M \quad I \tag{3}$$

where the operator $\sqsubseteq_M$ stands for modular abstraction[2]. In (2) $P(id, 2)$ is a generic process, i.e., a process with a symbolic value (id) as an id. This check is equivalent to $\forall j \in 2..N : (P(j, 2) \sqsubseteq_M I)$. Note that after the *enabling control abstraction* step the observable state space is finite and verifying (2) and (3) can be done automatically (by using model checking). More on this abstraction method can be found in the literature [16, 12, 14]. After the step of control abstraction we verify that $(P(1, 2) \| I) \models \varphi$ using model-checking. If $(P(1, 2) \| I)$ satisfies φ, then $\mathcal{Q}(N) \models \varphi$ for all N.

In this paper we focus on the second step that enables control abstraction. In the next section we describe the proposed abstraction technique in detail.

4. REDUCING OBSERVABLE STATE SPACE

In this section we describe the technique for reducing observable state space to a fixed finite set. This reduction enables the method of network invariants, i.e., control abstraction, to abstract the state space of N processes.

The correctness property φ does not have to be a persistence property. However, φ must be an LTL property that is expressed over variables in $V_g \cup V_{L(1)} \cup \{\mathtt{sv}[1]\}$ and in which the only temporal operators are $\Box$ and $\Diamond$. Note that for self-stabilizing systems the preprocessing step produces this type of correctness property and, therefore, enables the application of the method described in this section.

Let $V_{\mathtt{sv}}$ be the set of the shared variables, whose owners are the processes $P(2, N)$ to $P(N, N)$, i.e.,

$$V_{\mathtt{sv}} \triangleq \{\mathtt{sv}[j] | j \in 2..N\}$$

The purpose of this step is to replace these variables with an abstract variable $\mathtt{sv}_a[2]$ (Figure 1). The idea behind this transformation is that because of the low-atomicity constraint ($\Lambda 1$) only one variable of $V_{\mathtt{sv}}$ can be read or written by any action. Therefore, before the action is executed only one shared variable and its value are important. Hence, instead of $N-1$ shared variables, we need only one ($\mathtt{sv}_a[2]$) by giving the system the ability to write the value of any local copy of the variables $V_{\mathtt{sv}}$ to it. In Figure 2 an example of the application of the transformation is shown. In the next section we formally define the transformation and prove its soundness.

4.1 Obtaining Abstract System

We denote the specification of the abstract system produced as $\mathcal{Q}_a(N) = \langle \Sigma_a, F_a, \mathcal{N}_a, L_a\rangle$ and the concrete system as $\mathcal{Q}(N) = \langle \Sigma, F, \mathcal{N}, L\rangle$. Below, we describe how each of the components of $\mathcal{Q}_a(N)$ can be obtained from the corresponding components of $\mathcal{Q}(N)$.

State space: The only change in the state space is that the shared variables $\mathtt{sv} \in [1..N \rightarrow \mathrm{FS}]$ are replaced by $\mathtt{sv}_a \in [1..2 \rightarrow \mathrm{FS}]$. Let Σ be expressed as $\Sigma = \Sigma_{n\mathtt{sv}} \times [1..N \rightarrow \mathrm{FS}]$, where $\Sigma_{n\mathtt{sv}}$ is the state space of all variables except $\mathtt{sv}$. Then we can formally define Σ_a as $\Sigma_a = \Sigma_{n\mathtt{sv}} \times [1..2 \rightarrow \mathrm{FS}]$. We denote $\mathtt{sv}_a$ the variables in $[1..2 \rightarrow \mathrm{FS}]$.

Next state relation: The next state relation $\mathcal{N}_a$ of $\mathcal{Q}_a(N)$ is defined by a new set of actions $\tilde{A}$. We derive $\tilde{A}$ from some newly defined actions and the $\mathcal{Q}(N)$ actions. The following steps describe how we obtain $\tilde{A}$, which initially is an empty set.

T0 For each value $j \in 2..N$, we define and add to $\tilde{A}$ an action α_j of the form

$$\alpha_j \triangleq \quad \wedge \quad sv_a[2]' = l[j] \\ \quad \wedge \quad \mathrm{UNCHANGED}\langle all\_other\_variables\rangle$$

where $l[j]$ is the local copy of the shared variable $\mathtt{sv}[j]$ that process $P(j, N)$ maintains. The precondition of α_j is True in all states and its effect is to change $\mathtt{sv}_a[2]$ to a new value in FS. All other variables remain unchanged. Each α_j action is owned by process $P(j, N)$.

T1 For any action $\alpha \in A$ that does not read or write any of the variables in $V_{\mathtt{sv}}$, we create $\tilde{\alpha}$ by replacing all conjuncts $\mathtt{sv}[j]' = \mathtt{sv}[j]$ for all $j \in 2..N$ with $\mathtt{sv}_a[2]' = \mathtt{sv}_a[2]$. Then $\tilde{A} := \tilde{A} \cup \{\tilde{\alpha}\}$.

T2 For any action $\alpha \in A$ that reads variable $\mathtt{sv}[j]$, with $\mathtt{sv}[j] \in V_{\mathtt{sv}}$, we replace all occurrences of $\mathtt{sv}[j]$ with $\mathtt{sv}_a[2]$. Moreover, we replace all conjuncts of the form $\mathtt{sv}[j]' = \mathtt{sv}[j]$ with $\mathtt{sv}_a[2]' = \mathtt{sv}_a[2]$ and obtain $\tilde{\alpha}$. Then we add $\tilde{\alpha}$ to the set of actions $\tilde{A}$, i.e., $\tilde{A} := \tilde{A} \cup \{\tilde{\alpha}\}$.

T3 For any action $\alpha \in A$ that writes to variable $\mathtt{sv}[j]$, with $\mathtt{sv}[j] \in V_{\mathtt{sv}}$, we replace all occurrences of $\mathtt{sv}[j]'$ with $\mathtt{sv}_a[2]'$. In addition, we remove all conjuncts of the form $\mathtt{sv}[k]' = \mathtt{sv}[k]$ for all $k \neq j$ and obtain $\tilde{\alpha}$. Then we add $\tilde{\alpha}$ to $\tilde{A}$, i.e., $\tilde{A} := \tilde{A} \cup \{\tilde{\alpha}\}$.

[2]If $A \sqsubseteq_M B$, then for any environment all observable computations of A are observable computations of B. The environment can change any of the variables A and B do not own [12].

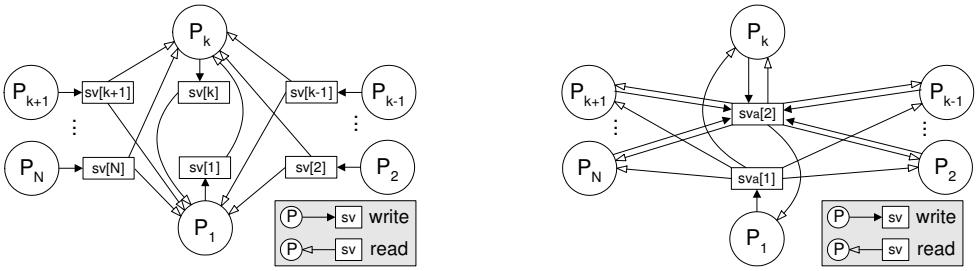

Figure 1: Left is the process graph before the transformation. Each process writes to its own shared variable and reads the shared variables of all other processes. For clarity the read edges of only $P(1,N)$ and $P(k,N)$ are shown. The processes do not need to read their own shared variable as they maintain a local copy of its value. The right part represents the system after the transformation. Only two shared variables exist in the system $\mathtt{sv}_a[1]$ and $\mathtt{sv}_a[2]$.

Steps T1–T3 are executed for $P(1,N)$ and $P(id,N)$[3] separately. The owners of the actions created by rules T1–T3 are the owners of the original actions. Note that there are no actions in A that read more than one element of $V_{\mathtt{sv}}$ or read and modify elements of $V_{\mathtt{sv}}$ because of restriction $\Lambda 1$. Consequently, any action in A is handled by one of the T1–T3 cases.

Initial states: The set of initial states F_a is formally defined by[4]

$$F_a \triangleq \{\, t_0 \mid \exists s_0 \in F_c : \;\; \wedge \;\; \Pi_{V_{nsv}\cup\{\mathtt{sv}[1]\}}(s_0) = \Pi_{V_{nsv}\cup\{\mathtt{sv}_a[1]\}}(t_0) \;\; \wedge \;\; \exists j \in 2..N : \Pi_{l[j]}(s_0) = \Pi_{\mathtt{sv}_a[2]}(t_0) \,\}$$

Set F_a is given by the projection of the initial states in F_c on the variables $V_{nsv} \cup \{\mathtt{sv}[1]\}$ and the variable $\mathtt{sv}_a[2]$ set equal to one of the local copies of the variables in $V_{\mathtt{sv}}$.

Liveness conditions: Based on the rule and actions used to define an action $\tilde{\alpha}$, its weak or strong fairness properties are specified. In some cases the same action $\tilde{\alpha}$ can be generated by more than one $\mathcal{Q}(N)$-actions using rule T2. Therefore, we consider each $\tilde{\alpha}$ to be constructed from a set of actions $A_s \subset A$. For any action $\tilde{\alpha}$ the fairness property added to L_a is the strongest property specified for any action in A_s. More formally, if $\tilde{\alpha}$ can be constructed from any $\alpha \in A_s$ using one of the rules T1–T3, then

$$\begin{aligned} A_s \cap \mathcal{S} \neq \emptyset \qquad\qquad & \Leftrightarrow \tilde{\alpha} \in \tilde{\mathcal{S}} \\ (A_s \cap \mathcal{W} \neq \emptyset) \;\wedge\; (A_s \cap \mathcal{S} = \emptyset) \;\; & \Leftrightarrow \tilde{\alpha} \in \tilde{\mathcal{W}} \end{aligned}$$

Besides the strong and weak fairness conditions on actions, we specify some liveness conditions related to constants. Let N_l be the minimum number of neighbors of process 1 and FS the domain of the shared variables. Then suppose that for any $N \geq N_l$ and for all behaviors of $\mathcal{Q}(N)$, there exists $k \in 2..N$ and $v_k \in$ FS, such that it holds $\Box(\mathtt{sv}[k] = v_k)$. If there exists an action $\alpha \in \mathcal{W}$, reading $\mathtt{sv}[k]$, then we define condition $c(e,e',v_k)$ obtained from $\langle \mathrm{eff}(\alpha)\rangle$ by replacing each occurrence of $\mathtt{sv}[k]$ with the value v_k. We define constraint

$$cf(\alpha) \;\triangleq\; \Box\Diamond\neg\mathrm{prec}(\alpha) \vee \Box\Diamond c(e,e',v_k)$$

[3]Process $P(id,N)$ is a generic process representing all processes $P(j,N)$ with $j \in 2..N$.

[4]Operator $\Pi_V(s)$ denotes projection of state s on the variables in V (Table 1).

For an action $\alpha \in \mathcal{S}$ accessing $\mathtt{sv}[k]$, the corresponding constraint will be

$$cf(\alpha) \;\triangleq\; \Diamond\Box\neg\mathrm{prec}(\alpha) \vee \Box\Diamond c(e,e',v_k)$$

Note that index k does not need to be the same for all behaviors. If the fairness properties are specified on a set of syntactically equivalent read actions that are defined for all $i \in 2..N$, the existence of a constant value in $V_{\mathtt{sv}}$ for all behaviors of $\mathcal{Q}(N)$ is sufficient for creating the constraint. We denote as $\mathcal{C}$ the set of the actions from which constant fairness conditions are generated. Then L_a can be expressed as

$$L_a = \bigwedge_{\tilde{\alpha}\in\tilde{\mathcal{W}}} wf(\tilde{\alpha}) \wedge \bigwedge_{\tilde{\alpha}\in\tilde{\mathcal{S}}} sf(\tilde{\alpha}) \wedge \bigwedge_{\alpha\in\mathcal{C}} cf(\alpha)$$

The property φ is not changed, because it is expressed over the variables in $V_g \cup V_{L(1)} \cup \{sv[1]\}$, which are present in both systems.

The following theorem states that the abstraction technique is sound.

THEOREM 1. *For any $N \geq 2$, if $\mathcal{Q}_a(N) \models \varphi$, then $\mathcal{Q}(N) \models \varphi$.*

PROOF SKETCH: The proof is based on the theory of refinement mappings [1]. We augment system $\mathcal{Q}(N)$ with a prophecy variable[5] $\pi \in 2..N$ to obtain system $\mathcal{Q}(N)^{\pi}$. The prophecy variable π holds the index of the next shared variable in $V_{\mathtt{sv}}$ that will be read or written. The initial value of π can be any element of $2..N$. We add one more action α_{π} that is always enabled and changes π to any of the values in $2..N$. Action α_{π} leaves all other variables of the system unchanged. All other actions of the new system do not modify π. Actions that read or modify variable $\mathtt{sv}[j] \in V_{\mathtt{sv}}$ are guarded by condition $\pi = j$. The new system $\mathcal{Q}(N)^{\pi}$ has equivalent liveness conditions to the liveness conditions

[5]A history variable is a variable that records past information and does not affect the behavior of the system. A prophecy variable is similar to a history variable, but instead of recording past information, it predicts future information. The system with the prophecy variables has at least as many behaviors as the original system. However, some non-deterministic decisions are made earlier and their result is recorded in the prophecy variable.

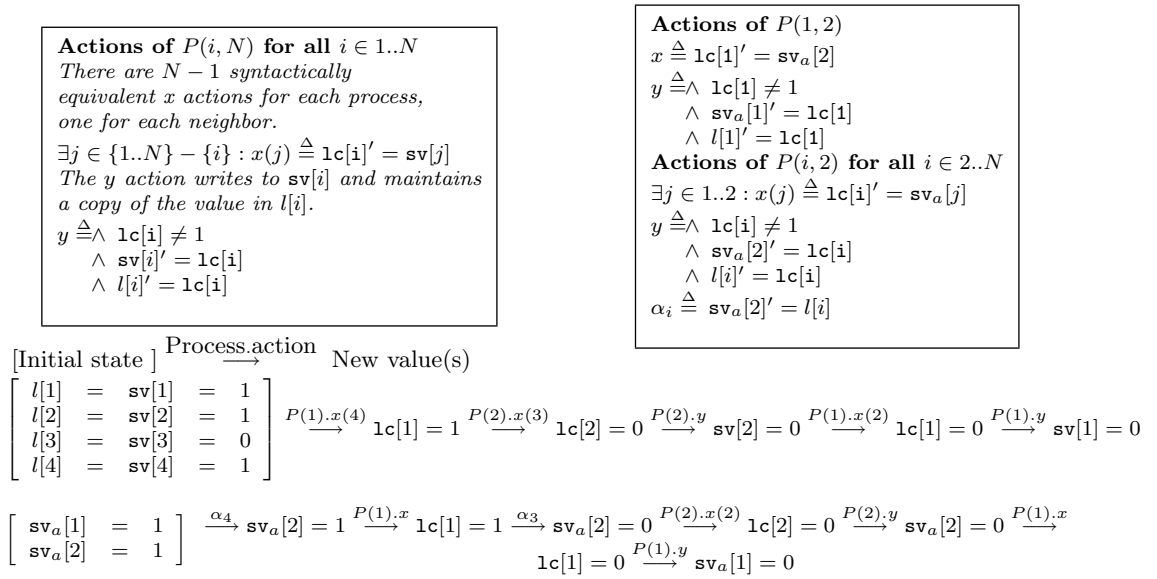

Figure 2: This is an example of the application of the transformation. Upper left figure shows the actions of $\mathcal{Q}(N)$. Upper right figure shows the actions of $\mathcal{Q}_a(N)$. For simplicity the part of each action specifying the variables left unchanged is not shown. For $P(id, N)$ actions x are transformed by rule T2, actions y by rule T3, and actions α_i are added because of rule T0. The upper sequence is a behavior segment of the original system. The bottom sequence is the corresponding behavior segment of the abstract system. The projections of two behavior segments over the variables in $V_{nsv} \cup \{\mathtt{sv}[1]\}$ are stuttering equivalent. Only the relevant values are displayed in the figure.

of $\mathcal{Q}(N)$. It is easy to prove then that $\mathcal{Q}(N)^\pi$ is a system obtained from $\mathcal{Q}(N)$ by adding a prophecy variable.

Then we define a refinement mapping f from the state space of $\mathcal{Q}(N)^\pi$ to the state space of $\mathcal{Q}_a(N)$. Let

$$s_c = (e, \mathtt{sv}[1], \mathtt{sv}[2], ..., \mathtt{sv}[N], \pi)$$

be a state of $\mathcal{Q}(N)^\pi$, where e is the projection of the state on V_{nsv}. Then

$$f(s_c) = (e, \mathtt{sv}_a[1], \mathtt{sv}_a[2])$$

where $\mathtt{sv}_a[1] = \mathtt{sv}[1]$ and $\mathtt{sv}_a[2] = \mathtt{sv}[\pi]$. In order for f to be a valid refinement mapping the following conditions need to be satisfied [1]:

R1. For each state s_c of $\mathcal{Q}(N)^\pi$: $\Pi_{V_{nsv} \cup \{\mathtt{sv}_a[1]\}}(f(s_c)) = \Pi_{V_{nsv} \cup \{\mathtt{sv}[1]\}}(s_c)$.

R2. $f(F^\pi) \subseteq F_a$, where F^π is the set of initial states of $\mathcal{Q}(N)^\pi$.

R3. If $\langle s_c, t_c \rangle \in \mathcal{N}^\pi$ then $\langle f(s_c), f(t_c) \rangle \in \mathcal{N}_a$ or $f(s_c) = f(t_c)$, where $\mathcal{N}^\pi$ is the next state relation of $\mathcal{Q}(N)^\pi$.

R4. $f(\mathcal{X}^\pi) \subseteq L_a$, where $\mathcal{X}^\pi$ is the set of computations specified by $\mathcal{Q}(N)^\pi$.

Properties R1 and R2 hold by construction of $\mathcal{Q}(N)^\pi$ and $\mathcal{Q}_a(N)$. For R3 we can show that the transition caused by an action β of $\mathcal{Q}(N)^\pi$, which originated from an action α of the original system, can be simulated in the abstract system by a transition using $\tilde{\alpha}$, which is the action created from α by rules (T1–T3). A transition caused by α_π can be simulated by the transition of an action α_j (T0 rule) in the abstract system. Property R4 is the hardest to prove. Let $f(\sigma^\pi) \not\models L_a$ be a sequence, such that $\sigma^\pi \models \mathcal{Q}(N)^\pi$. We show that this can lead to a contradiction. Sequence $f(\sigma^\pi)$ must violate a weak, strong, or constant fairness property of L_a. Assume $f(\sigma^\pi)$ violates the weak fairness property of action $\tilde{\alpha}$, then there exists action α in the original system, from which $\tilde{\alpha}$ is obtained, such that $\sigma^\pi \models \Box\Diamond\neg\text{prec}(\alpha) \vee \Box\Diamond\langle\text{eff}(\alpha)\rangle$. If $\sigma^\pi \models \Box\Diamond\neg\text{prec}(\alpha)$, then we know $f(\sigma^\pi) \models \Box\Diamond\neg\text{prec}(\tilde{\alpha})$, since α and $\tilde{\alpha}$ have the same precondition which is expressed over variables in V_{nsv} only (assumption Λ3). Therefore, it must hold that $\sigma^\pi \models \Box\Diamond\langle\text{eff}(\alpha)\rangle$. If action α does not read or modify a variable in $V_{\mathtt{sv}}$, we know that $f(\sigma^\pi) \models \Box\Diamond\langle\text{eff}(\tilde{\alpha})\rangle$. Consequently, α must read or modify a shared variable. Then using assumption Λ4, we can show that $\langle\text{eff}(\alpha)\rangle$ is infinitely often satisfied because α or a syntactically equivalent action is executed infinitely often. In either case we have that $f(\sigma^\pi) \models \Box\Diamond\langle\text{eff}(\tilde{\alpha})\rangle$, which leads to a contradiction. Similarly, we can prove that strong and constant fairness conditions of L_a are satisfied by any sequence $f(\sigma^\pi)$ with σ^π a behavior of $\mathcal{Q}(N)^\pi$. □

Next section describes sufficient conditions for the completeness of the abstraction technique.

4.2 Completeness Conditions

In this subsection we present the conditions under which the proposed abstraction technique is complete.

An important concept is that of syntactically equivalent actions. Suppose we have two actions α and β of $\mathcal{Q}(N)$, then $\alpha \equiv_{se} \beta$ if and only if α and β belong to the same process

and β can be obtained from α by replacing every instance of $\mathtt{sv}[j]$ with $\mathtt{sv}[k]$, where j and k are constants in $2..N$. The relation $\equiv_{se}$ is an equivalence relation. All actions of the same equivalence class are transformed to one action by our technique.

The following conditions are sufficient for completeness:

C1 Every reachable state of the system is an initial state.

C2 The read actions of the system form equivalence classes, such that the size of each equivalence class is not bounded from above as N increases.

C3 The number of processes in the system is not bounded from above.

C4 For all N_1 and N_2 with $2 \leq N_1 < N_2$, the systems $\mathcal{Q}(N_1)$ and $\mathcal{Q}(N_2)$ have the same formula as liveness constraint, i.e., the liveness constraint is independent of N.

C5 No read action has a fairness constraint.

If $\mathcal{Q}(N)$ is a self-stabilizing system, then C1 is always true. Conditions C2 and C3 are commonly satisfied by uniform parameterized systems, in which no process distinguishes a finite number of its neighbors as special processes. Many distributed algorithms that read the values of all neighbors in a loop, which is not an atomic action, satisfy condition C5. Finally, in many cases condition C4 is satisfied as well. This is because the environment of process 1 is normally not constrained by fairness conditions. A fairness constraint on the environment, i.e., an expectation that the environment at some point in the future will make a move if some conditions are satisfied, can sometimes be added to the condition $\Diamond\Box H$ of the original condition property ($\Diamond\Box H \rightarrow \Diamond\Box J$). Therefore, by increasing the convergence steps we can remove some fairness requirements for the environment.

THEOREM 2. *If C1–C5 hold and* $\mathcal{Q}_a(N) \not\models \Diamond\Box J$, *then* $\exists K \geq N : \mathcal{Q}(K) \not\models \Diamond\Box J$.

PROOF. The idea of the proof is to construct a counterexample for some instance of the concrete system using the counterexample of the abstract system. Since the correctness property is a liveness property, the abstract counterexample is a lasso-shaped sequence, i.e., a cycle and a path leading from an initial state to a state in the cycle. This cycle represents the infinite part of the counterexample. However, in our case the correctness property is $\Diamond\Box J$, and every state is an initial state (C1). Therefore, we can consider only the cycle in the counterexample, which satisfies $\Box\Diamond\neg J$. Using this cycle we can produce a cycle in the concrete state space, which is a behavior of $\mathcal{Q}(K)$ for some $K \geq N$.

First, we determine the number of processes K in the concrete system. Then using induction we show how we can create a concrete counterexample from the abstract counterexample.

To determine the number K of processes we consider the reads in the cycle of the abstract counterexample. There could be a read action $\tilde{\alpha}$ performed by a process j in the abstract counterexample, such that for any k in the equivalence class of that action α in the concrete system $\mathtt{sv}_a[2] \neq \mathtt{sv}[k]$, i.e., the value, which $\mathtt{sv}_a[2]$ has, is not equal to any value of the shared variables that process j could read by executing action α. Because of condition C3 we can add more processes in the system; and due to C2, we know that by increasing the number of processes, we can add at least one new process m to the equivalence class of the read action of j. Since any reachable state is an initial state and because the processes are uniform, we choose $\mathtt{sv}[m] = \mathtt{sv}_a[2]$ and a corresponding valid local state for m. Consequently, we start from a state s_0 in the abstract cycle and for each transition (s_i, s_{i+1}), we determine whether the action $\tilde{\alpha}$ that caused the transition is a read action. In case $\tilde{\alpha}$ is a read action, let $[\alpha]_{se}$ be the equivalence class of actions from which $\tilde{\alpha}$ was obtained. We make sure that there is at least one process j with $l[j] = \mathtt{sv}_a[2]$ in s_i, where $l[j]$ is the local copy of $\mathtt{sv}[j]$, and one action of $[\alpha]_{se}$ reading $\mathtt{sv}[j]$. We repeat this process until we check all transitions of the cycle. We denote the number of processes in the system after the procedure as K. The added processes do not perform any action and, therefore, their local states and owned shared variables maintain the same values. Consequently, we still have a cycle in the extended state space. Moreover, because of C4 there are no fairness constraints added that could be violated.

The second step of the proof is to build the concrete counterexample for system $\mathcal{Q}(K)$ from the abstract counterexample. We start again from state s_0. We create state t_0 by assigning to the local variables, i.e., $\bigcup_{i \in 1..K} V_{L(i)}$, of the K processes the same values as in s_0. We do the same for the global variables in V_g. For the shared variables we assign the values of the local copies, i.e., $\forall j \in 1..N : \mathtt{sv}[j] = l[j]$. For every transition (s_i, s_{i+1}) of the abstract system caused by $\tilde{\alpha}$, we create transition (t_i, t_{i+1}) in the concrete system by executing action α from which $\tilde{\alpha}$ is obtained by rules T1–T3[6]. If $\tilde{\alpha}$ is obtained from rule T0, we execute the stuttering step. The action α determined this way has the same effect on the local and global variables as $\tilde{\alpha}$. For the shared variables the effect is the same as the effect on the local copies. The preconditions of the actions do not depend on the shared variables and, therefore, if $\tilde{\alpha}$ is enabled in s_i, α is enabled in t_i.

Since property J is expressed over the variables in $V_{nsv} \cup \{\mathtt{sv}[1]\}$ and the projections of the two counterexamples on these variables are stuttering equivalent sequences, the concrete cycle satisfies $\Box\Diamond\neg J$. Actions used in rules T1 and T3 have single corresponding actions in abstract system that have the same fairness conditions. Therefore, their fairness conditions are satisfied. Moreover, because of condition C5 the fairness conditions of the read actions cannot be violated. □

Conditions C1–C5 are sufficient to prove completeness, but not all of them are necessary. For example, C2–C4 are not needed if the equivalent classes of the read actions include all elements of $\mathtt{sv}[1], ..., \mathtt{sv}[N]$. In such a case we do not need to extend the number of processes of the system.

[6]If $\tilde{\alpha}$ can be obtained from multiple syntactically equivalent read actions, we choose one of them such that the shared variable $\mathtt{sv}[j]$ read by the chosen action is equal to the $\mathtt{sv}_a[2]$ value in state s_i. Because of the extension of the system from N to K processes such an action always exists.

5. CASE STUDIES

In this section we demonstrate the effectiveness of our technique on 3 self-stabilizing algorithms: leader-election, coloring, and spanning-tree construction. These three problems are often encountered during the design of distributed embedded systems [3, 19, 15]. We specify the algorithms and apply our technique using TLA+ [17]. Then we use the TLC model checker [17], which is based on explicit state enumeration, to prove the correctness of the algorithm. All 3 examples satisfy the conditions for soundness after the preprocessing step. The conditions for completeness (C1–C5) are not necessary, but sufficient for the automated creation of the concrete counterexample. The examples presented in the paper do not satisfy C4–C5. Therefore, we cannot exclude the possibility of a spurious counterexample. However, no counterexample was produced in any of the case studies. Proving correctness in all 3 examples demonstrates the effectiveness of our approach as a sound abstraction technique.

5.1 Leader Election

For the leader election algorithm ([7],p35) a number of processes form an arbitrary connected graph. The purpose of the algorithm is that eventually all processes will agree that the process with the minimum id in the graph is the leader. In order to achieve that, each process stores a candidate leader and its distance from the leader. Then it reads the values of all its neighbors. If there is a candidate with id smaller than its own leader or with the same id and smaller distance, the process updates its candidate with that value and its distance by incrementing the read distance by 1. The update happens only if the distance of the neighbor's candidate is less than a prespecified constant M, which represents the maximum number of nodes in the graph.

If one of the processes i is initialized with a candidate id v in variable `leader` which is smaller than any of the ids in the graph and any of the other nodes' candidates, then v is stored in other neighbors' `leader` variable and from them again to i. However, each time this "floating" id moves from one node to another, the value of `distance` increases. Therefore, eventually `distance` becomes greater than M for all nodes and the value v does not appear in the `leader` variables of the graph.

For this algorithm we assume that `min_id` is the smallest floating id and `min_dist` is its minimum `distance` value in the graph and we prove that eventually always if a node has `min_id` as a candidate, the distance will be greater than `min_dist`. We use $P(1, N)$ as the special process, but due to symmetry the proof can be generalized.

This system is not a finite state system because variables storing candidate leaders and distances take values from an infinite domain. Therefore, we abstract those variables to a few values of interest, i.e., $\{\texttt{min\_id}, \texttt{other\_id}\}$ for the candidates and $\{\texttt{min\_dist}, \texttt{gt\_min\_dist}, \texttt{any\_dist}\}$ for the distance, where `gt_min_dist` denotes greater than min distance. We abstract the part of the graph that does not belong to process 1 and its neighbors and the loop structure that reads the values of all neighbors. Moreover, we simplify the correctness condition to $\Diamond\Box J$, where

$$J = \wedge\ \texttt{candidate}[1] = \texttt{min\_id} \Rightarrow \texttt{distance}[1] = \texttt{gt\_min\_dist}$$
$$\wedge\ \texttt{leader}[1] = \texttt{min\_id} \Rightarrow \texttt{dis}[1] = \texttt{gt\_min\_dist}$$

After the preprocessing step we are left with a parameterized system that is amenable to our technique. The domain for each process is finite. Each process executes a low-atomicity algorithm with single-writer, multi-reader shared variables. Moreover, the preconditions of the actions depend on the program counter or on local variables. The usage of the program counter makes the system satisfy $\Lambda 4$, as discussed in Section 2. The shared variable for this algorithm is $\texttt{sv}[j] = (\texttt{leader}[j], \texttt{dis}[j])$. After the application of our technique the system has a fixed number of observable variables and is amenable to control abstraction. By performing control abstraction we obtain a finite-state system with 500,000 states. It took TLC 22 minutes to prove the correctness of $\Box\Diamond J$.

5.2 Coloring

We apply our technique on the self-stabilizing coloring algorithm ([7],p162). The purpose of this algorithm is to assign a color to each process, such that no two neighbor processes have the same color. Each process keeps reading the values of all its neighbors, stores the values of the neighbors with an id greater than its id, and assigns to its `color` variable a color that none of the neighbors with a higher id have.

We want to prove that eventually always a process will have a color that none of its neighbors with higher ids have, if eventually always the neighbors with higher ids are silent processes, i.e., the values of their shared variables are constant. After the preprocessing step we apply our technique and then control abstraction. TLC requires 1 minute to prove the desired property with the total number of states being 26,496.

In one of the actions of the program, a process makes a call to a function `choose`() with arguments the set of all colors except the colors of its neighbors with higher ids. If no fault occurs or after a process has completed its first actions, it is guaranteed that the argument passed to `choose`() has at least one element. This is because the number of colors is always greater than the number of neighbors of a node. However, if we start in a state in which the two sets are equal, the argument passed to `choose`() is the empty set. Therefore, it is important that `choose`() is able to return a color if called with the empty set and not crash. In the paper [8] from which the paragraph in [7] is motivated, the authors mention that.

5.3 Spanning-Tree Construction

The last example to which we applied our abstraction technique is Arora's and Gouda's low-atomicity spanning-tree algorithm [2]. In this algorithm, each node stores the root of the tree and its distance from the root. We assume that floating root ids have been eliminated, the nodes that are in distance $l - 1$ from the root have stabilized, and all other nodes that have identified the root have distance values greater or equal to l. Then, using our abstraction technique and control abstraction, we prove with TLC that the

node $P(1, N)$ at distance l stabilizes. Due to symmetry we can conclude that all nodes in distance l stabilize. It takes 2 minutes for TLC to prove the property and the total number of states of the final abstract system is 32,000.

6. CONCLUSIONS

In this paper we have presented an abstraction technique that enables the use of control abstraction for low-atomicity, shared-memory self-stabilizing systems that do not have a fixed number of observable variables for each process. The technique can be used as part of an abstraction methodology that includes a preprocessing step and control abstraction. Although we have developed abstraction techniques that facilitate the preprocessing and control abstraction steps for self-stabilizing systems, these techniques are beyond the scope of this paper. We consider the proposed abstraction technique to be a critical step in an abstraction methodology for the verification of parameterized self-stabilizing systems, because it is required if control abstraction is to be applied to the examples we have studied in this paper.

7. REFERENCES

[1] Abadi, M., and Lamport, L. The existence of refinement mappings. *Theoretical Computer Science 82*, 2 (1991).

[2] Arora, A., and Gouda, M. Distributed reset. *IEEE Transactions on Computers 43*, 9 (1994).

[3] Arumugam, M., and Kulkarni, S. S. Self-stabilizing deterministic TDMA for sensor networks. *Distributed Computing and Internet Technology* (2005), 69–81.

[4] Baukus, K., Lakhnech, Y., and Stahl, K. Verification of parameterized protocols. *Journal of Universal Computer Science 7*, 2 (2001), 141–158.

[5] Burns, J. E., Gouda, M. G., and Miller, R. E. Stabilization and pseudo-stabilization. *Distributed Computing 7*, 1 (1993), 35–42.

[6] Clarke, E., Talupur, M., and Veith, H. Environment abstraction for parameterized verification. In *VMCAI 2006: Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation* (London, UK, 2006), Springer-Verlag, pp. 126–141.

[7] Dolev, S. *Self-Stabilization*. The MIT Press, 2000.

[8] Dolev, S., and Herman, T. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science 1997*, 4 (December 1997).

[9] Emerson, E. A., and Kahlon, V. Reducing model checking of the many to the few. In *CADE-17: Proceedings of the 17th International Conference on Automated Deduction* (London, UK, 2000), Springer-Verlag, pp. 236–254.

[10] Fang, Y., Piterman, N., Pnueli, A., and Zuck, L. Liveness with invisible ranking. *International Journal on Software Tools for Technology Transfer (STTT) 8*, 3 (2006), 261–279.

[11] Gouda, M. G., and Multari, N. J. Stabilizing communication protocols. *IEEE Transactions on Computers 40*, 4 (1991), 448–458.

[12] Kesten, Y., and Pnueli, A. Control and data abstraction: the cornerstones of practical formal verification. *International Journal on Software Tools for Technology Transfer (STTT) 2*, 4 (2000).

[13] Kesten, Y., and Pnueli, A. Verification by augmented finitary abstraction. *Information and Computation 163*, 1 (2000), 203–243.

[14] Kesten, Y., Pnueli, A., Shahar, E., and Zuck, L. D. Network invariants in action. In *CONCUR '02: Proceedings of the International Conference on Concurrency Theory* (London, UK, 2002), Springer-Verlag, pp. 101–115.

[15] Kulkarni, S. S., and Arumugam, U. Collision-free communication in sensor networks. In *Self-Stabilizing Systems: 6th International Symposium, SSS 2003, San Francisco, CA, USA, June 24-25, 2003. Proceedings* (2003), Springer Berlin / Heidelberg.

[16] Kurshan, R. P., and McMillan, K. L. A structural induction theorem for processes. *Information and Computation 117*, 1 (1995), 1–11.

[17] Lamport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2002.

[18] Lynch, N. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[19] Malpani, N., Welch, J. L., and Vaidya, N. Leader election algorithms for mobile ad hoc networks. In *DIALM '00: Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications* (New York, NY, USA, 2000), ACM, pp. 96–103.

[20] Pnueli, A., Xu, J., and Zuck, L. Liveness with (0,1,infinity)-counter abstraction. In *CAV '02: Proceedings of the International Conference on Computer Aided Verification* (London, UK, 2002), Springer-Verlag, pp. 107–122.

[21] Whittlesey-Harris, R. S., and Nesterenko, M. Fault-tolerance verification of the fluids and combustion facility of the international space station. In *ICDCSW '06: Proceedings of the 26th IEEE International ConferenceWorkshops on Distributed Computing Systems* (Washington, DC, USA, 2006), IEEE Computer Society, p. 5.