Incremental Component-Based Modeling, Verification, and Performance Evaluation of Distributed Reset *

Ananda Basu Borzoo Bonakdarpour Marius Bozga Joseph Sifakis

VERIMAG
Centre Équation
2 ave de Vignate, 38610
Gières, France

Abstract

Design and implementation of distributed algorithms often involve many subtleties due to their complex structure, nondeterminism, and low atomicity as well as occurrence of unanticipated physical events such as faults. Thus, constructing correct distributed systems has always been a challenge and often subject to serious errors.

We present a methodology for incremental and component-based modeling, verification, and performance evaluation of a self-stabilizing algorithm known as distributed reset*. The methodology is based on the BIP component framework. In BIP, a system is modeled as the composition of a set of atomic components by using two types of operators: interactions describing synchronization constraints between components, and priorities to specify scheduling constraints. The methodology involves three steps. First, a high-level model of the algorithm is built in BIP from the set of its processes by using powerful primitives for multiparty interactions and scheduling. Then, we use this model for the verification of functional properties including closure, deadlock-freedom, and finite reachability of the set of legitimate states. Finally, a distributed model which is observationally equivalent to the high-level model is generated. This model is used for performance analysis taking into account the degree of parallelism and convergence times for failure-free behavior as well as in the presence of faults. We choose* distributed reset *for our case study due to its simplicity and elegance. The original algorithm works in the shared memory model and we demonstrate how refinement of a small set of very simple guarded commands to a less high-level model involves many subtleties that may dramatically affect the correctness of the refined model and how BIP facilitates the process of rigorous modeling, simulation, and verification.*

We believe that this work opens the path for further research on component-based formal modeling, verification, and deployment of more complex distributed systems.

Keywords: Component-based specification, Verification, Self-stabilization, Distributed algorithms, Reset algorithms, Modeling.

1 Introduction

Distributed systems are constructed from a set of relatively independent components that form a unified, but geographically and functionally diverse entity. They remain difficult to design, build, and maintain, because of their inherently concurrent, nondeterministic, and non-atomic structure as well as the occurrence of unanticipated physical events such as faults.

We currently lack disciplined methods for rigorous design and correct implementation of distributed systems. These systems are still being constructed in an ad-hoc fashion in practice, mainly for two reasons: (1) formal methods are not easy to use by designers and developers; and (2) there is a wide gap between modeling formalisms and automated verification tools on one side, and practical development and deployment tools on the other side. It is not clear how existing results can be consistently integrated in design and implementation methodologies.

Formalisms such as process algebras [1], I/O automata [21], and UNITY [9] have been used for modeling and reasoning about the correctness of distributed systems. Numerous techniques and algorithms have also been introduced for adding reliability and fault-tolerance to distributed systems. Moreover, an interest has recently emerged in verification of distributed algorithms. For instance, model checking protocols in the promising area

*This manuscript is the full version of the brief announcement short paper published at DISC'09. This work is sponsored by the COMBEST European project. For all correspondence, please contact Borzoo Bonakdarpour at borzoo@imag.fr or +33 (0)4 56 52 04 22.

of transactional memory has received considerable attention [11, 16–18]. Verification of agreement algorithms has also been studied from different perspectives. For instance, in [27], the authors use bounded model checking [10] to verify the correctness of consensus algorithms. While these approaches play an important role in formalizing and achieving correctness of distributed algorithms, we believe that a more systematic and unified approach for modeling, verification, and as importantly deployment of distributed systems in a natural and practical fashion (multi-threaded, component-based, and incremental) is still required.

In this paper, we apply a methodology which consistently integrates modeling, verification, and performance evaluation techniques, based on the BIP (Behavior, Interaction, Priority) component framework developed at Verimag [3, 4]. BIP is based on a semantic model encompassing composition of heterogeneous components. In contrast to all other formalisms using a single type interaction (e.g., rendezvous, asynchronous message passing), BIP uses two families of composition operators for expressing coordination between components: *interactions* and *priorities*. Interactions are expressed by combining two protocols: rendezvous and broadcast. In [7], we have proposed a notion of expressiveness for component-based systems and have shown that BIP is more expressive than any formalism based on a single type of interaction. Supporting tools of BIP's theory include techniques for model verification [24] as well as for generating from a high-level component-based model in BIP an observationally equivalent multi-threaded distributed implementation [3].

BIP is implemented in a toolset which, (1) provides an expressive formal specification language [7] yet easy-to-understand by engineers and developers; (2) proposes a fully disciplined methodology for incremental and component-based architectural and hierarchical development; (3) provides state exploration tools for model checking; and (4) generates executable multi-threaded C++ code out of a formally specified model. BIP has successfully been used for modeling and verifying complex robotics as well as real-time multimedia systems [4, 5].

To illustrate our methodology, we have chosen the self-stabilizing distributed reset algorithm due to Arora and Gouda [2]. Our choice is not due to efficiency or practicality of the algorithm, but to its simplicity and elegance. The distributed reset algorithm consists of two layers: a tree layer for constructing a global rooted spanning tree across processes and a wave layer for resetting the entire system through a diffusing computation mechanism. In the original algorithm, each layer is consists of a small set of very simple guarded commands in the shared memory model. Pioneered by Dijkstra [12], a self-stabilizing distributed algorithm guarantees that starting from an arbitrary state, it converges to a legitimate state (from where it satisfies its specification) and remains in a legitimate set of states thereafter. Nevertheless, as Dijkstra points out in a belated proof of correctness of his token ring algorithm [13], designing and deploying correct self-stabilizing algorithms is not a trivial task at all, although it initially seems straightforward. In the context of distributed reset, we demonstrate how refinement of such a simple algorithm to a less high-level model involves many subtleties that may dramatically affect the correctness of the refined model. We also show how BIP facilitates rigorous modeling, verification, and performance analysis of the distributed reset algorithm. In particular, our methodology involves three steps:

- The starting point is a high-level BIP model of a distributed system obtained as the composition of a set of components representing its processes. This model represents a system with a global state and atomic transitions. Multiparty interactions (interactions that involve an arbitrary number of components) may lead the system from one state to another. Modeling a distributed system in such a high-level model confers numerous advantages such as modularity and faithfulness as coordination is directly expressed by using protocols instead of low-level primitives. We show how BIP allows independent modeling of the tree layer and wave layer of distributed reset as the composition of atomic components. Composition involves in addition to interactions, scheduling constraints expressed as dynamic priorities among interactions.

- We use this compact high-level model for verification of functional (implementation independent) properties. We specify safety and liveness properties that any self-stabilizing algorithm must satisfy. These properties include *closure*, *deadlock-freedom*, and *finite reachability* of the set of legitimate states. We verify these properties on our BIP model for distributed reset by using model checking techniques.

- A multi-threaded (distributed) executable C++ model is generated from the high-level model. This C++ model faithfully represents an actual distributed implementation of the high-level model. In other words, the logical properties and dynamics of the C++ model conforms with the high-level model and an actual distributed C++ implementation. The generation is fully automated. The multi-threaded (distributed) model is obtained by applying two transformations preserving observational equivalence, following results in [3]: (1) multiparty interactions are substituted by protocols based on asynchronous message passing; (2) the state of a component is undefined (due to distribution) when it performs some internal computation. The multi-threaded model which meets all the functional properties of the global-state model, is used for estimating performance of the implementation. This includes analysis taking into account the degree of parallelism and convergence times for failure-free behavior as well as in the presence of faults.

Organization of the paper. In Section 2, we review the distributed reset algorithm and basic concepts of the BIP framework. In Section 3, we formally model the distributed reset algorithm in the BIP language. Section 4 is dedicated to verification of distributed reset using model checking. We describe our experiments and analyze the performance of distributed reset in Section 5. Finally, we discuss future work and present concluding remarks in Section 6.

2 Background

In this section, we present an intuitive overview of our case study and the methodology used in order to incrementally model, verify, and evaluate the performance of distributed algorithms. In Subsection 2.1, we present the distributed reset algorithm [2]. Then, in Subsection 2.2, we present an overview of the BIP framework [4, 26].

2.1 Distributed Reset

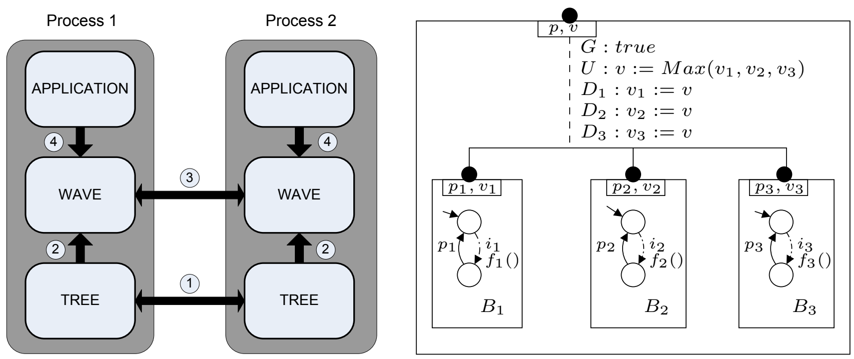
Intuitively, distributed reset augments functionality of a distributed system with a subsystem where each process can initiate a global reset to a predefined global state. Each process is associated with a set of adjacent processes with which it can communicate. At any time instant, an alive process may crash which results in change of the list of adjacent processes. The reset subsystem consists of three layers where each layer is embedded in each process as an independent local component:

- In the tree layer, adjacent processes communicate in order to construct and maintain a rooted spanning tree throughout the alive processes. Thus, any changes in the adjacency relationship of processes eventually result in corresponding changes in the structure of the spanning tree. The tree layer is self-stabilizing in the sense that starting from any arbitrary topology (e.g., a grid, ring, etc) and initial structure (e.g., a forest of rooted trees, cyclic graphs, etc), construction of a rooted spanning tree within a finite number of steps is guaranteed. In other words, faults such as process failures and local variable corruptions do not result in permanent destruction of the spanning tree.
- The application layer may locally choose to initiate a global reset. In this case, the corresponding local component sends a request to the local wave layer described next.
- The wave layer may receive a reset request from the application layer in order to start a global reset. In this case, the local wave component of a process forwards the request to its parent in the current spanning tree until the request reaches the root. Once the root receives a reset request, it initiates a *diffusing computation* as follows. First, the root resets its own state and then initiates a *reset wave*. The reset wave travels towards the leaves of the spanning tree and causes the wave component of each encountered process to reset its state. When the reset wave reaches a leaf process, it bounces as a *completion wave* that travels towards the root process. A process propagates the completion wave to its parent if all its offsprings are complete. When the completion wave reaches the root, the reset is complete. In the algorithm proposed by Arora and Gouda [2], each wave component maintains a *session number* in order to ensure that concurrent resets do not interfere. The wave layer is also self-stabilizing in the sense that starting from any arbitrary configuration of the wave components, the algorithm guarantees an eventual global reset within a finite number of states.

Figure 1(a) illustrates the overall architecture of two adjacent processes in distributed reset. The communication channels between these two processes are as follows:

1. Adjacent tree components communicate to construct and maintain a spanning tree (Channel 1).
2. A tree component communicates with the local wave component to indicate changes in parent-child relationship of the process caused by modifications in the structure of the spanning tree (Channel 2).
3. Wave components that are in parent-child relationship communicate in order to accomplish a global reset (Channel 3).
4. An application component may send reset requests to the local wave component (Channel 4).

Notice that the wave layer always assumes the existence of a sound rooted spanning tree. Thus, the only piece of information that a tree component shares with the corresponding local wave component is the identity of the parent process in the spanning tree.



(a) The overall architecture of two adjacent processes in distributed reset.

(b) A simple BIP example.

Figure 1: Preliminary concepts.

2.2 The BIP Framework

In the BIP language [4,6,26], an architecture is characterized as a hierarchically structured set of *components* obtained by composition from a set of atomic components. Composition is parameterized by sets of *interactions* between the composed components. The BIP toolset has a compilation chain allowing the generation of C++ code from BIP models. The generated code is modular and can be executed on a dedicated middleware consisting of an Engine which orchestrates the computation of atomic components by executing their interactions. Hierarchical description allows incremental reasoning and progressive design of complex systems. *Priorities* among interactions allow specifying scheduling policies in BIP.

A BIP component is characterized by its *interface* and its *behavior*. An interface consists of a set of *external* ports used to specify interactions. Each port p has some associated variables v_p which are visible when an interaction involving p is executed. It is assumed that the ports and associated variables of atomic components are disjoint.

The behavior of atomic components is described as a finite state automaton extended with data and functions given in C++. A transition of the automaton is labeled with a trigger and a function f describing a local computation. A trigger consists of a guard g on (local) data and a port p through which an interaction is sought. For a given control state, a transition can be executed if its guard g is true and an interaction involving p is possible (we precisely define the notion of interactions later in this section). Execution of transitions is atomic: it is initiated by the interaction and followed by the execution of f. We emphasize that a component may have *internal* ports as well. Transitions labeled by internal ports are executed independently and do not require initiation of an interaction.

Graphical notation. An atomic component (i.e., its behavior and interface) is placed in a box (see Figure 1(b) for an example). Each external port and its corresponding variables are placed in a rectangle inside its containing component. Behavior of a component is described by the classic notation of an automaton; a hollow circle denotes a control state and an arrow denotes a transition. For the sake of clarity, we use a solid (respectively, dotted) arrow to denote a transition triggered by an external (respectively, internal) port.

Composition consists in applying a set of *connectors* to a set of components. A connector is defined by:

1. an *exported port* p and the associated variables;

2. its *support set of ports* $\{p_1, \ldots, p_n\}$ of the composed components;

3. its set of *interactions*, that are, subsets of the set $\{p_1, \ldots, p_n\}$. Each interaction $\alpha = \{p_{i_1} \ldots p_{i_k}\}$ is annotated by

   (a) a *guard* G, boolean expression involving variables associated with the ports p_{i_j} involved in the interaction α;

   (b) an *upstream transfer function* U specifying flow of data from variables associated with the support set of ports towards the associated variables of the exported port;

   (c) and *downstream transfer functions* $D_{i_1}, \ldots, D_{i_k}$ specifying flow of data from the variables associated with the exported port towards variables associated with the support set of ports.

When it is clear from the context, we simply denote a connector by only its support set of ports (i.e., $\langle p_1 \dots p_n \rangle$). The set of interactions associated with a connector is defined using a typing mechanism of ports in its support set of ports. We distinguish two types of ports: *synchron* and *trigger*. Any set of support ports that is either maximal or it contains a trigger denotes a valid interaction. Intuitively, a synchron is a passive port, and needs synchronization with other ports. In other words, such a port cannot initiate an interaction without synchronizing with other ports. However, a special case (such as the one in Figure 1(b)) is a connector that only involves synchrons. Such a connector denotes a *rendezvous* and requires all ports to participate. On the other hand, a trigger is an active port, and can initiate an interaction without synchronizing with other ports.

The global behavior resulting from the application of a connector to a set of components is defined as follows. An interaction $\alpha = \{p_{i_1} \dots p_{i_k}\}$ of the connector is enabled only if for each one of its ports p_{i_j}, there exists an enabled transition in some component labeled by p_{i_j}. Execution of the interaction involves two steps:

1. a temporary variable v is assigned the value $U(v_{p_{i_1}}, \dots, v_{p_{i_k}})$;
2. the variables v_{i_j} associated with the ports p_{i_j} are assigned values $D_{i_j}(v)$.

The execution of an interaction is followed by the execution of the local computations of the synchronized transitions.

Graphical notation (cont'd). A connector is represented as a solid line connecting all ports in its support set. The exported port by a connector is placed over the connector. A solid circle attached to an external port denotes a synchron and a triangle denotes a trigger (see Figure 2(b) for an example).

A *composite component* is recursively obtained from a set of atomic or sub-components by successive (i.e., acyclic) application of connectors. The support set of any connector contains ports exported either by subcomponents or other existing connectors.

Graphical notation (cont'd). A composite component is also respresented as a box containting its subcomponents and their respective connector hierarchy.

Example. In Figure 1(b), we provide a simple composite component. It is composed of three atomic components B_1, B_2, and B_3. Each atomic component B_k holds an integer variable v_k, exported through an external port p_k. Additionally, the component has an internal port i_k which triggers the execution of an internal computation defined by the function f_k. The ternary connector defines the interaction $\{p_1, p_2, p_3\}$ which is a rendezvous among external ports p_1, p_2, and p_3. As a result of this interaction, following the definition of upstream an downstream transfer functions, each component receives the maximum of the exported values. Moreover, notice that the exported port of the connector belongs to the interface of the composite component, that is, it can be used for further interactions.

3 Modeling Distributed Reset in BIP

We model distributed reset according to the BIP system construction methodology: (1) designing the *behavior* of each atomic component (i.e., an automaton extended by variables and ports), (2) applying synchronization mechanisms for ensuring coordination of components through *interactions*, and (3) specifying scheduling constraints by using *priorities*.

We first model the wave layer and the tree layer of the algorithm independently in Subsections 3.1 and 3.2, respectively, by applying the above methodology to each layer. Then, we add cross-layer connectors in order to interconnect the two independent layers in Subsection 3.3. From the wave and tree components designed in this section, one can incrementally build a distributed system equipped with the distributed reset functionality according to a topology of interest.

3.1 The Wave Layer

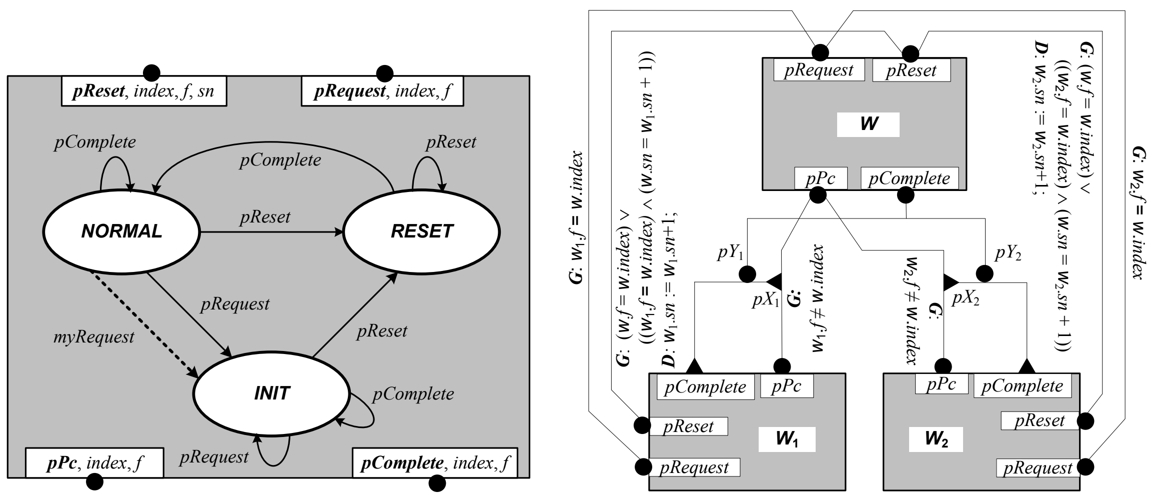
The wave layer of distributed reset assumes that a perfect rooted spanning tree exists throughout the distributed system. Thus, the wave layer is only concerned with achieving a self-stabilizing diffusing computation to accomplish a distributed reset. Each process in the distributed system contains a *wave atomic component*. We describe the wave atomic component in terms of normal, faulty, and recovery behaviors of the wave layer.

3.1.1 Normal Operation

We start with modeling the normal operation of the wave layer, where each component works perfectly in the absence of faults.

Interface and Behavior

- *(Exported Ports)* A wave component has the following four ports: (1) $pRequest$ for propagating a reset request from a child to its parent, (2) $pReset$ for enforcing a child to reset its state by the parent, (3) $pComplete$ for informing a node that its subtree



(a) Behavior

(b) Interactions

Figure 2: Normal operation of the wave layer.

has completed diffusing computation, and (4) pPc for identifying adjacent processes that are neither a child nor a parent. As can be seen in Figure 2(a), each port is associated with a subset of variables of the component.

- *(Variables)* Each component maintains the following variables: (1) an integer $index$ to represent the unique index of the component, (2) an integer f to keep the index of the parent process in the spanning tree, and (3) an integer sn for the session number of the current ongoing reset.

- *(Automaton)* A wave component has three *control states*: *NORMAL*, *INIT*, and *RESET* (see Figure 2(a)). Initially, all components are in the *NORMAL* control state. A wave component may move to *INIT* by either enabling the $myRequest$ internal port (e.g., from the application layer of the same process) or when a reset request is received via the $pRequest$ port. This move occurs during the *request wave*. Next, the component moves from *INIT* to *RESET* and resets its state when the port $pReset$ is enabled during the *reset wave*. A component may also move from *INIT* to *RESET* on port $pReset$, if it was not involved in the request wave. Finally, a wave component moves back to *NORMAL* on port $pComplete$, when its subtree has completed the completion wave. A completed wave component is either in *NORMAL* control state or in *INIT* if another reset has already been initiated in its subtree. The $pComplete$ self-loop at this control state is added for this reason.

Interactions

Interactions among wave components are specified in terms of a set of connectors between them. Notice that each process is associated with a set of adjacent processes according to a topology. However, not all adjacent processes are normally in parent-child relationship and, hence, are not allowed to communicate. In order to make our design as flexible as possible, the static design of connectors should provide the potential of communication between any two adjacent processes depending upon the topology. Nonetheless, the actual communication in the wave layer should occur only between processes that are allowed to do so. This is similar to designing a circuit of electronic components with a set of switches, where depending upon the state of switches only a subset of components are actually involved in the circuit. In this context, let w be a wave component whose adjacent neighbors are $w_1 \cdots w_n$. We categorize the interactions based on the three waves of the wave layer:

- *(Request Wave)* The first set of connectors is $\{\langle (w.pRequest)(w_i.pRequest)\rangle \mid 1 \leq i \leq n\}$. These connectors allow the component w at *INIT* to synchronize with a component w_i, that is already in control state *INIT*: w_i synchronizes with w by taking the $pRequest$ self-loop at control state *INIT*. Figure 2(b) presents an example, where w has two adjacent processes w_1 and w_2[1]. Observe that the connec-

[1]The connectors involved in Figure 2(b) essentially construct Channel 3 of Figure 1(a).

tors between $pRequest$ ports are associated with a guard to ensure correct parent-child relationship and bottom-up flow of requests (e.g., $w.index = w_1.f$). Hence, if two processes are adjacent due to the topology, but not in any parent-child relationship, they do not interact to send or receive reset requests. This guard is present in almost all of the connectors in the wave layer [2]. Recall that since BIP allows us to associate ports with variables, evaluation of the above guard does not require explicit use of shared memory.

- *(Reset wave)* The second set of connectors is $\{\langle(w.pReset)(w_i.pReset)\rangle \mid 1 \leq i \leq n\}$. Once the root (of the spanning tree) wave component moves to *INIT*, it goes to *RESET* without synchronizing on port $pReset$. This is managed through specifying a guard on this type of connectors. That is, the corresponding synchronization is bypassed, if $w.f = w.index$ (see Figure 2(b)). Once a non-root process is in *RESET*, its children can go to *RESET* from either *NORMAL* or *INIT* by synchronizing on port $pReset$. In other words, a child whose parent is in *RESET* can reset its state regardless of its past desire to initiate a global reset. A parent synchronizes with its resetting children through the $pReset$ self-loop at control state *RESET*. Similar to the connector between $pRequest$ ports, we ensure that only a parent can propagate the reset wave to its children by specifying a guard on the connector between $pReset$ ports. This guard also ensures that the session number of a child is one less than the session number of its parent. Finally, when the reset connector gets enabled, it increments the session number of the child component, to mark the number of the current reset wave.

- *(Completion wave)* The design of connectors for the completion wave is a bit more complex, as a process declares completion only if all its children are complete (which essentially means its entire subtree is complete). The completion mechanism inherently requires a multi-party rendezvous. However, our design should be flexible in the sense that it allows bypassing irrelevant adjacent processes as well as synchronizing with real children processes. To this end, we construct a hierarchical connector as follows. First, we include a connector between pPc ports of w and w_i, where $1 \leq i \leq n$ (see Figure 2(b)). This connector gets enabled when w and w_i are *not* in a parent-child relationship. The connector exports a port called pX_i, which gets triggered when the completion of w_i is irrelevant to w. Now, the pair of pX_i and $w_i.pComplete$ constructs another connector, which exports the port pY_i. This port is present in the rendezvous that covers all w_i components. The full interaction can be characterized by the following rendezvous: $\langle(w.pComplete)pY_1 pY_2 \cdots pY_n\rangle$, where $pY_i = \langle(pX_i) + (w_i.pComplete)\rangle$ and $pX_i = \langle(w.pPc)(w_i.pPc)\rangle$. The '+' operator denotes a choice between two enabled ports. As an example, if w is a leaf in the spanning tree, it does not wait for any of the adjacent processes to complete, as pX_i is active for all i.

Notice that starting from an initial state and operating normally, the global state of the set of all components in the wave layer arranged on a rooted spanning tree should remain in the following set of *legitimate states*:

$$
\begin{aligned}
\mathcal{S}_w \equiv \forall w_1, w_2 \ :: & \\
& ((w_1.f = w_2.index \wedge \neg\, w_2.\mathit{RESET}) \ \Rightarrow \\
& \qquad (\neg w_1.\mathit{RESET} \wedge w_1.sn = w_2.sn)) \ \wedge \\
& ((w_1.f = w_2.index \wedge \ \ w_2.\mathit{RESET}) \ \Rightarrow \\
& \qquad ((\neg w_1.\mathit{RESET} \wedge w_2.sn = w_1.sn + 1) \vee \\
& \qquad\qquad w_2.sn = w_1.sn)).
\end{aligned}
$$

where w_1 and w_2 are two wave components.

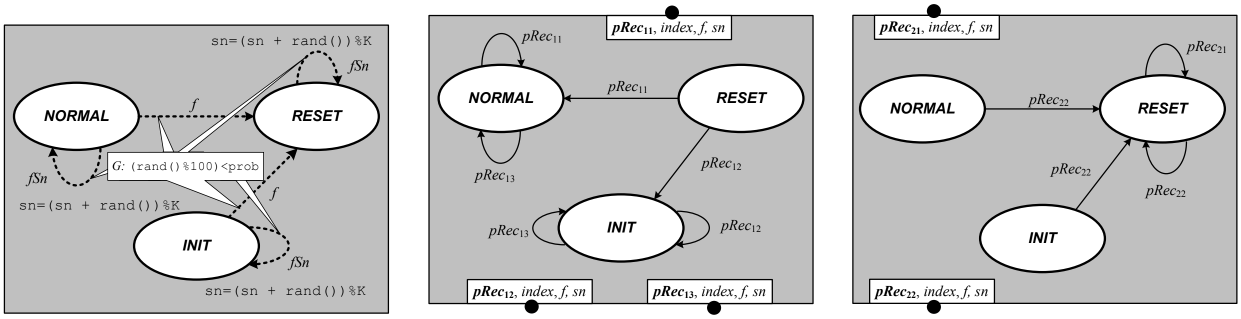
3.1.2 Faulty Behavior

In distributed reset, faults can lead a process to reach any arbitrary state. To capture the notion of faults, it suffices to focus on transitions that reach a state in $\neg\mathcal{S}_w$. These faults are modeled in Figure 3(a). The transitions labeled by internal port f cause a process to go to *RESET* from either *INIT* or *NORMAL* without synchronizing with its parent. Faults labeled by *fSn* are self-loops that corrupt the session number of a process by executing the C assignment `sn = (sn + rand()) % K`, where `K` is the maximum number of processes. To make the occurrence of faults a random event, we associate the guard of fault transitions with a probability `prob`. Notice that the union of transitions in Figures 2(a) and 3(a) may lead a wave component to reach any arbitrary state. Finally, we emphasize that we do not associate any synchronization with faults. This allows faults to occur under no synchronization constraints.

3.1.3 Self-Stabilization (Recovery)

Interface and Behavior. We model self-stabilization of the wave layer based on the two conjuncts of $\mathcal{S}_w$. Essentially, the recovery mechanism should ensure that starting from any state in $\neg\mathcal{S}_w$, the entire distributed system can reach a state in $\mathcal{S}_w$ within a finite number of steps. To

[2] We note that symmetric conditions should be added to the guard of connectors to cover all cases among adjacent processes (e.g., w_1 is parent of w). We omit them in the figure for simplicity.



(a) Faulty behavior

(b) Recovery type 1

(c) Recovery type 2

Figure 3: Self-stabilization of the wave layer.

this end, we add the following behavior to each wave component. First, we consider the case where a parent process is not in *RESET*, but one of its children is. To resolve this case, it suffices for the child to (1) move to the control state where its parent is (i.e., either *NORMAL* through synchronization on port $pRec_{11}$ or *INIT* through port $pRec_{12}$), and (2) copy the session number from the parent to maintain consistency (see Figure 3(b)). To resolve the case where a parent and its child are in the same control state but their session numbers differ, the processes synchronize on port $pRec_{13}$ and the child copies the parent's session number (see Figure 3(b)).

The second type of recovery behavior resolves the cases where the second conjunct of $\mathcal{S}_w$ is violated. In particular, if a process and one of its children are in *RESET*, but their session numbers differ, then they synchronize on port $pRec_{21}$ and the child copies the session number. Finally, if a process is in *RESET*, but one of its children is not in *RESET* and the child's session number is not one less than its parent's, then they synchronize on port $pRec_{22}$ and the child copies the session number.

Interactions. Recovery connectors define interactions on corresponding ports between adjacent components. Thus, the set of connectors is $\{\langle(w.pRec_{jk})(w_i.pRec_{jk})\rangle \mid (i = 1..n) \wedge (j = 1..2) \wedge (k = 1..3)\}$, where w_i is adjacent to w. Similar to the normal operation, we associate guards with recovery connectors to ensure the correct parent-child relationship among the adjacent processes. Moreover, we incorporate data transfer in interactions for copying session numbers.

3.2 The Tree Layer

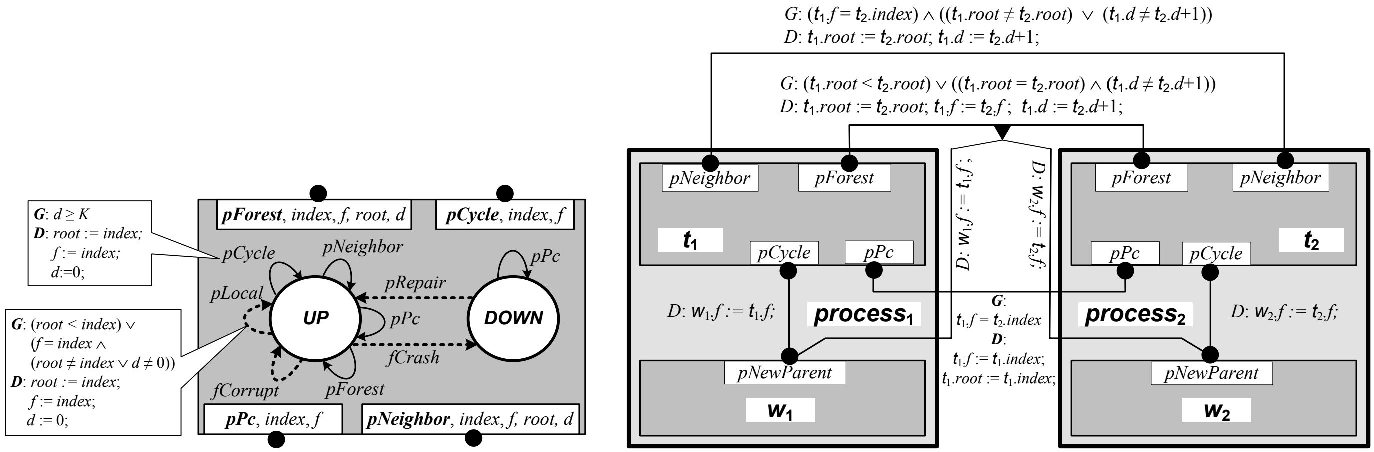
Unlike the wave layer, the tree layer is only concerned with a self-stabilizing algorithm for constructing a rooted spanning tree.

Interface and Behavior

- *(Exported Ports)* Adjacent processes in the tree layer communicate via three ports: (1) $pForest$ when two adjacent processes identify two different roots, (2) $pNeighbor$ when two adjacent processes identify an *inconsistency* between them (i.e., different roots, incorrect shortest distance to the root, or a root process that has a parent), and (3) pPc when a parent process crashes (see Figure 4(a)). Port $pCycle$ is used for cross-layer interactions described in Subsection 3.3.

- *(Variables)* Each tree component maintains the following variables: (1) an integer $index$ to represent the unique index of the component, (2) an integer f to keep the index of the parent process in the spanning tree, (3) an integer $root$ that contains the index of the root process, and an integer d whose value is the distance of the process to the root. The value of $index$ is equal to that of the corresponding wave component and is specified statically. The value of f, however, is determined at runtime across the tree layer. Thus, the tree and wave components of a process need to communicate to maintain consistency. We address this issue in Subsection 3.3. Each component also maintains an array N, which contains the index of all adjacent processes.

- *(Automaton)* Initially, all processes are alive and in the *UP* control state (see Figure 4(a)). Faults can alter the value of variables f, $root$, and d arbitrarily through the internal port $fCorrupt$. Also, each process may crash and go to the control state *DOWN* through the internal port $fCrash$. A crashed process may get repaired and return to the *UP* control state



(a) Tree component

(b) The tree layer and cross-layer interactions

Figure 4: The tree layer.

through internal port $pRepair$. Thus, faults can potentially break a rooted tree into forests, create cycles, and cause (local or global) inconsistencies. A tree component participates in resolving the above issues when it is in control state *UP*. A local inconsistency is detected in a tree component through the internal port $pLocal$ associated with a guard which indicates a discrepancy in the value of either $root$ or d (see Figure 4(a) for the inconsistency conditions). A cycle can also be detected locally, if the distance of a process to the root is greater than the maximum number of processes K. A tree component fixes a local inconsistency and breaks a cycle by setting $root = f = index$ and $d = 0$ (see Figure 4(a)).

Interactions

Similar to the wave layer, interactions among tree components are specified in terms of a set of connectors between them according to a topology. Let t be a tree component whose adjacent processes are $t_1..t_n$. The interactions between tree components resolve the following issues to construct a rooted spanning tree:

- *(Process crashes)* The set $\{\langle(t.pPc)(t_i.pPc)\rangle \mid 1 \leq i \leq n\}$ of connectors are used to inform a process that its parent has crashed. As can be seen in Figure 4, this connector is enabled when one participating component is in *UP* and the other process is in *DOWN* control state. The guard of the connector enforces the parent-child relationship. Execution of this interaction invalidates the variables of the child process whose parent is crashed. Recall that interactions between tree components construct Channel 1 of Figure 1(a).

- *(Parental inconsistencies)* A connector in the set $\{\langle(t.pNeighbor)(t_i.pNeighbor)\rangle \mid 1 \leq i \leq n\}$ is enabled when a child and its parent either do not agree on the same root, or, the child is not located one step farther of its parent from the root. In either case, the child simply fixes the root index and its distance according to the parent through the data transfer mechanism of the connector (see the guard G and transfer function D of the connector in Figure 4(b)).

- *(Rooted forests)* A connector in the set $\{\langle(t.pForest)(t_i.pForest)\rangle \mid 1 \leq i \leq n\}$ is enabled when multiple roots are detected by a tree component. This situation occurs when there exists an adjacent process whose root has a higher index or the process offers a shorter distance to the root. In this case, the process updates its $root$, f, and d variables via the data transfer mechanism (see the guard G and function D of the connector in Figure 4(b)).

Finally, we define the set of legitimate states of the tree layer, where a rooted tree that spans over all alive processes exists, as follows:

$$
\begin{aligned}
\mathcal{S}_t \equiv\ & (k = \max\{t.index \mid t.\mathit{UP}\}) \wedge \\
& (\forall t_1 \mid t_1.\mathit{UP}:: \\
& \quad (t_1.index = k \quad \Rightarrow \\
& \qquad (t_1.index = t_1.root \wedge
\end{aligned}
$$

$$
\begin{aligned}
&\quad t_1.index = t_1.f \ \wedge \ t_1.d = 0)) \wedge \\
&(t_1.index \neq k \quad \Rightarrow \\
&\quad (\exists t_2 \in t_1.N :: (t_1.f = t_2.index \wedge \\
&\qquad t_1.d = t_2.d + 1 \wedge \\
&\qquad\quad \forall t_3 \in t_1.N :: t_2.d \leq t_3.d)))).
\end{aligned}
$$

3.3 Building Distributed Reset

Given the tree layer and wave layer components, one can easily compose them and incrementally build a distributed reset system. To this end, we add cross layer interactions as follows. When a cycle or multiple forests are detected in the tree layer, a tree component may choose a new parent from its neighbors. In this case, the wave component of the same process has to update its parent as well, so the subsequent resets complete maturely (see Channel 2 in Figure 1(a)). Thus, we augment each wave component with a $pNewParent$ port, which synchronizes with $pCycle$ or an exported port by the $pForest$ connectors to update its parent (see Figure 4(b)).

4 Model Checking Distributed Reset

We have verified the distributed reset algorithm using classic model-checking. For a finite instantiation of the distributed reset algorithm (a grid topology folded on a sphere), we started by constructing a finite representation of its overall behavior as a flat labeled transition systems (LTS) using BIP state-space explorer [4]. States correspond to configurations reached by the distributed reset algorithm, and transitions taken to move from one configuration to another are labeled by the interactions introduced in Section 3. On the LTS model, we have evaluated a set of temporal logic formulas encoding the key properties of the distributed reset algorithm, using the EVALUATOR tool of CADP [15, 22].

We express the properties using a generic characterization of interactions (i.e., labels). We note that given the set of legitimate states, such labeling can be easily automated in the context of verification of self-stabilizing algorithms:

- We add a self-loop labeled *steady* to each legitimate state. For the wave layer (respectively, tree layer), all these self-loops participate in a global rendezvous interaction whose guard satisfies expression $\mathcal{S}_w$ (respectively, $\mathcal{S}_t$) introduced in Section 3.

- We label each internal fault transition introduced in Section 3 by *fault*. This labeling makes the occurrence of a fault an observable event.

- We label the remaining interactions by *prog*. This includes recovery as well as interactions that participate in constructing a spanning tree at the tree layer and interactions that contribute in achieving a global reset at the wave layer.

We provide the exact definition of properties in *regular alternation-free μ-calculus* which is the temporal logic formalism handled by the EVALUATOR tool. This logic is an extension of the alternation-free μ-calculus [20] with action formulae as in ACTL [23] and regular expressions over action sequences as in PDL [14]. The full syntax and semantics can be found in [22]. We consider the following properties that any self-stabilizing system must satisfy:

- *(closure)* legitimate states are preserved by taking non-fault actions (only faults may reach an illegitimate state from a legitimate state):
  $\phi_1 : [any^*] \, (\langle steady \rangle \mathbf{T} \Rightarrow [prog] \langle steady \rangle \mathbf{T})$[3]

- *(deadlock-freedom)* from any reachable state, there exists an outgoing program transition:
  $\phi_2 : [any^*] \langle prog \rangle \mathbf{T}$

- *(reachability)* starting from any state, a legitimate state can be reached by taking only program actions (there always exist a path from any state to a legitimate state):
  $\phi_3 : [any^*] \langle prog^* \rangle \langle steady \rangle \mathbf{T}$

- *(convergence)* starting from any state, a legitimate state is *eventually* reached by taking only program actions (the algorithm never reaches a cycle outside legitimate states):
  $\phi_4 : [any^*] \neg \nu X. \, (\neg \langle steady \rangle \mathbf{T} \wedge \langle prog \rangle X)$

In order to reduce the complexity of verification of distributed reset, we utilize a compositional approach. Specifically, we infer the correctness of the composite distributed reset algorithm by verifying the correctness of the tree layer and wave layer individually. However, such compositional verification needs demonstration of *interference-freedom* between components. Let C_1 and C_2 be two components. We say that C_1 and C_2 do not interfer with each other if whenever C_1 satisfies some property ϕ_1 and C_2 satisfies some property ϕ_2, then their "composition" (e.g., using BIP interactions) satisfies $\phi_1 \wedge \phi_2$.

[3] We recall that $q \models \langle a \rangle \varphi$ iff $\exists q \xrightarrow{a} q' : q' \models \varphi$, where q and q' are two states, $\xrightarrow{a}$ is a transition labeled by a, and φ is a formula. Also, $q \models [a]\varphi$ iff $\forall q \xrightarrow{a} q' : q' \models \varphi$. The label *any* denotes any transition label, i.e., $\{steady, prog, fault\}$, $\mathbf{T}$ denotes logical true, and $*$ denotes a sequence. Finally, ν and μ respectively denote the largest and smallest fixpoints in the μ-calculus.

	n	states	transitions	generation time	ϕ_1	ϕ_2	ϕ_3	ϕ_4
tree	4	56	649	< 1	< 1	< 1	< 1	< 1
	6	7022	81390	29	1	1	2	3
	9	2456936	59409357	4000	10	23	19	145
wave	4	996	5840	< 1	< 1	< 1	< 1	< 1
	6	27590	189523	36	2	2	3	5
	9	1539001	7077649	2500	5	7	6	93

Table 1: Verifying distributed reset using classic model checking.

Theorem 1. The composition of the tree layer and wave layer in the distributed reset algorithm is interference-free for properties $\phi_1 ... \phi_4$.

Proof. Notice that the only interaction between the tree layer and wave layer occurs when a change of parent is decided by the tree layer. This interaction only involves a unilateral change of parent at the wave layer by the tree layer. Thus, the wave layer does not interfere with the tree layer in any way. Moreover, when the wave layer is silent, a change of parent does not change the state of the wave layer. Thus, the only possible pitfall of the aforementioned interaction is where an ongoing reset at the wave layer coincides with a change of parent at the tree layer. Since there exist only a finite number of actions at the tree layer to construct a spanning tree, the wave layer eventually complete its execution on the current spanning tree as well. The only consequence of a change of parent is that the ongoing reset completes immaturely, which is a known and permitted phenomenon in the original algorithm as well. ■

The immediate consequence of Theorem 1 is that separate verification of the layers results in the correctness of distributed reset. In order to generate LTS models of manageable size for a reasonably large number of processes in the algorithm we manually applied the following well-known model checking heuristics on the BIP model:

- We apply *abstraction* by reducing the domain of values of each variable to the minimal possible set. For instance, when a fault alters the value of the *root* variable in a process, the exact new value does not matter and, hence, the corresponding illegitimate state can be encoded by a single *corrupted value* for the *root* variable.

- We perform a *live analysis* [8] in every component and based on it, we re-initialize each variable as soon as it becomes dead on a computation path.

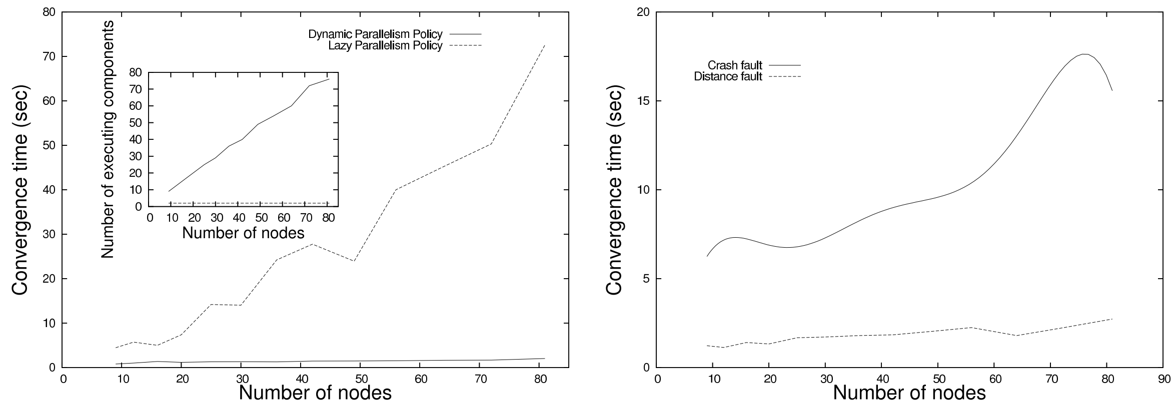
- Finally, we *simplify the sequence* of occurrence of faults by allowing multiple types of faults occurring at the same time.

Table 1 summarizes the results about the size of the models in terms of number of processes in the grid. The LTS generation time as well as the time needed to verify the properties considered are all in seconds. All verification tasks are run on a PC with a 3.2GHz Intel Xeon processor and 4GB RAM.

5 Performance Evaluation

The BIP toolset provides us with means for generating C++ multi-threaded code from high-level BIP models. This feature enables us to evaluate the performance of distributed algorithms described by high-level models. This allows in particular, to evaluate the impact of changes to the high-level model without getting involved with its actual C++ implementation. We emphasize that the logical properties and dynamics of the C++ model conform with the high-level model and an actual C++ implementation. Below, we present the result of some of our experiments and lessons learned in evaluating the performance of distributed reset. All experiments in this section are run on a PC with a Pentium IV 3GHz processor and 1GB memory under Debian Linux. All plots on each graph is the average value of 10 runs for the corresponding experiment. The reason for this number of experiments is due to the fact that our models do not exhibit a high level non-determinism. In fact, we observed that the result of experiments do not fluctuate significantly.

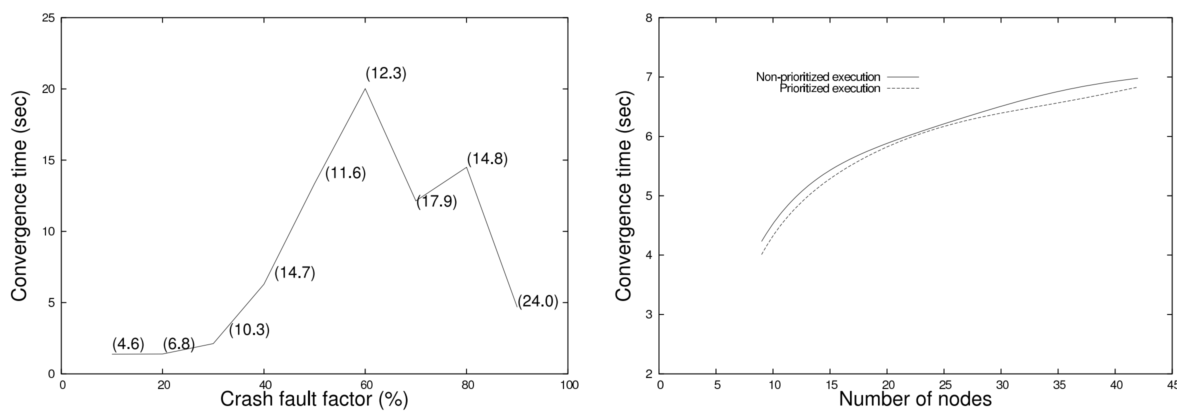
Degree of Parallelism. The BIP *Engine* uses different parallelism policies to execute distributed models. In a *lazy* policy, the Engine executes only one interaction at a time. In other words, it waits for all atomic components to complete their internal computation before initiating a new interaction. Conversely, in a *dynamic* policy, the Engine allows multiple interactions to be executed in parallel as long as the overall execution conforms with the sequential semantics (i.e., their executions are observationally equivalent). Figure 5(a) compares the convergence time of the tree layer under these policies in the absence



(a) Degree of parallelism

(b) Crash fault vs. variable corruption

Figure 5: Performance analysis of the tree layer.



(a) Effect of fault factor.

(b) Effect of priorities.

Figure 6: Performance analysis of the tree layer and wave layer.

of faults. As can be seen, the graph shows that under the dynamic policy convergence is much faster, as the Engine allows multiple tree components to work simultaneously. This makes performance evaluation of distributed algorithms very close to reality.

Severity of Faults. Figure 5(b) compares the effect of d-variable corruption and crash faults on convergence time of the tree layer. The graphs clearly show that crash faults' damage to the spanning tree is more severe than the case where a process has wrong coordinates of the root. This result is expected, as crashing a node requires reconstructing the spanning, which can be costly. For instance, if a crashed process is the root, the entire spanning tree has to be reconstructed. On the other hand, a d-variable corruption can be fixed by a single interaction with one of the adjacent processes.

Figure 6(a) shows the behavior of the tree layer in the presence of crash faults, where the probability of occurrence of such faults decreases by a *fault factor* ff, where $\mathit{ff} < 1$. That is, if the current probability of a crash is p for a process, after the process is repaired, the probability of the subsequent crash for this process is $\mathit{ff} * p$. As can be seen, the convergence time increases as the fault factor grows to 60%. However, when the fault factor grows beyond 60%, the tree layer converges faster. This is because there are so many crashed processes that are not repaired and, hence, not participating in forming a spanning tree. Thus, a high fault factor reduces the size of actual distributed system (the average number of process crashes for some plots are available in Figure 6(a)).

Effect of specifying priorities. Figure 6(b) shows the effect of granting priority to execution of tree layer over the wave layer. The idea is when the spanning is broken, the algorithm should focus on reconstructing a new tree rather than letting the wave layer work. In fact, simultaneous operation of both layers may result in completing immature resets. In BIP, one can easily specify priorities among interactions. In particular, we specify a local priority for the tree layer interactions of adjacent processes and Figure 6(b) shows slightly faster convergence for the prioritized tree layer.

6 Conclusion

The paper illustrates the application of a methodology consistently integrating high-level modeling and verification of functional properties with performance analysis of a distributed implementation. Consistency is ensured by results guaranteeing preservation of properties of the initial high-level model by its implementation.

We demonstrated how one can build-up the self-stabilizing distributed reset algorithm attributed to Arora and Gouda [2] by developing a set of independent atomic components and then wiring them by using connectors. We also identified a set of safety and liveness properties that any self-stabilizing algorithm has to satisfy. These properties include *closure*, *deadlock-freedom*, and *finite reachability* of the set of legitimate states starting from any arbitrary state. We successfully verified these properties for each layer of distributed reset for a grid topology

BIP allows a natural high-level description of the coordination between atomic components by using structured connectors and multiparty interactions. Modeling the same coordination with formalisms based on point-to-point interaction is a non-trivial problem. It would have required the use of additional atomic components implementing multiparty interactions by protocols. The obtained models can be modified incrementally. New processes can be added or deleted without disturbing the operation of the system.

Furthermore, the high-level model abstracts from silent actions used to implement multiparty interactions in the distributed model [3]. This drastically simplifies property verification.

For performance evaluation, we used a distributed model functionally equivalent to the high-level BIP model. This model can be used for implementation purposes. It is based on operational semantics which allows a rigorous analysis of extra-functional properties. The obtained benchmarks show the effect of scheduling policies and of different types of faults on convergence times and the degree of parallelism. Here again incremental description by adding or removing architectural features has been very useful for modifying the model.

We believe that our approach can be used for modeling, verifying, and evaluating distributed algorithms. It advantageously combines an expressive and rigorous high-level component-based formalism and its associated distributed implementation, which is beneficial for more complex algorithms such as concurrency control techniques. In this context, we are currently working on a generic component-based framework for modeling and analyzing transactional memory [19, 25] algorithms using BIP. We are also working on a wide range of transformations from high-level BIP models into low-level actual implementations such as the Message Passing Interface (MPI), multi-core, and fully distributed platforms. Another interesting research direction is to automate the procedure presented in this paper by transforming algorithms in (shared memory) guarded commands into BIP models.

References

[1] M. Alexander and W. Gardner. *Process Algebra for Parallel and Distributed Processing*. Chapman & Hall/CRC, 2008.

[2] A. Arora and M. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:316–331, 1994.

[3] A. Basu, P. Bidinger, M. Bozga, and J. Sifakis. Distributed semantics and implementation for systems with interaction and priority. In *Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 116–133, 2008.

[4] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 3–12, 2006.

[5] A. Basu, M. Gallien, C. Lesire, T. H. Nguyen, S. Bensalem, F. Ingrand, and J. Sifakis. Incremental component-based construction and verification of a robotic system. In *European Conference on Artificial Intelligence (ECAI)*, pages 631–635, 2008.

[6] S. Bliudze and J. Sifakis. Causal semantics for the algebra of connectors. In *Formal Methods for Components and Objects (FMCO)*, pages 179–199, 2007.

[7] S. Bliudze and J. Sifakis. A notion of glue expressiveness for component-based systems. In *Concurrency Theory (CONCUR)*, pages 508–522, 2008.

[8] M. Bozga, J.-C. Fernandez, and L. Ghirvu. State-space reduction based on live variable analysis. *Journal of Science of Computer Programming*, 47(2-3):203–220, 2003.

[9] K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[10] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[11] A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck. Verifying correctness of transactional memories. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 37–44, 2007.

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

[13] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.

[14] M. J. Fischer and R. E. Ladner. Propositional Dynamic Logic of Regular Programs. *Journal of Computer and System Sciences*, 18:194–211, 1979.

[15] H. Garavel, F. Lang, R. Mateescu, and W. Serve. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV)*, pages 158–163, 2007.

[16] R. Guerraoui, T. A. Henzinger, B. Jobstmann, and V. Singh. Model checking transactional memories. In *Programming Language Design and Implementation (PLDI)*, pages 372–382, 2008.

[17] R. Guerraoui, T. A. Henzinger, and V. Singh. Completeness and nondeterminism in model checking transactional memories. In *Concurrency Theory (CONCUR)*, pages 21–35, 2008.

[18] R. Guerraoui, T. A. Henzinger, and V. Singh. Software transactional memory on relaxed memory models. In *Computer Aided Verification (CAV)*, 2009, To appear.

[19] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[20] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[21] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[22] R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, 2003.

[23] R. D. Nicola and F. Vaandrager. Action versus State Based Logics for Transition Systems. In *Semantics of Systems of Concurrent Processes (La Roche Posay, France)*, pages 407–419, 1990.

[24] T. N. S. Bensalem, M. Bozga and J. Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *Computer Aided Verification (CAV)*, pages 614–619, 2009.

[25] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[26] J. Sifakis. A framework for component-based construction extended abstract. In *IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 293–300, 2005.

[27] T. Tsuchiya and A. Schiper. Using bounded model checking to verify consensus algorithms. In *Distributed Computing (DISC)*, pages 466–480, 2008.