Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 149 (2006) 49–60
www.elsevier.com/locate/entcs

Automatic Verification of Fault-Tolerant Register Emulations

Paul C. Attie[1]

College of Computer Science, Northeastern University, Boston, MA, USA.
and
MIT CSAIL, Cambridge, MA, USA.

Hana Chockler[2]

IBM Research Laboratory, Haifa, Israel.

Abstract

The design and verification of fault-tolerant distributed algorithms is a complicated task. Usually, the proof of correctness is done manually, and thus depends on the skill of the prover. Using automated verification methods, such as model checking, can greatly simplify the verification. However, model checking of distributed algorithms is often intractable because of the state-explosion problem. In this paper we present a novel approach to verification of quorum-based distributed register emulation algorithms with undetectable crash failures of processes. Our approach is based on projection and abstraction and allows us to reduce the task of model-checking the whole system to fair model-checking of subsystems consisting of a constant number of processes. Our method is highly scalable and can be applied to a large class of algorithms. Aside from efficient verification, it can also be used for finding redundancies in existing algorithms.

Keywords: Distributed algorithms, parametrized systems, fault-tolerance, crash failures, register emulation, automatic verification, model checking

1 Introduction

Formal verification is widely recognized as a key means for assuring the correct behavior of large and complex software systems. Such systems are usually

[1] Email: attie@ccs.neu.edu
[2] Email: hanac@il.ibm.com

doi:10.1016/j.entcs.2005.11.016

distributed, and contain a large number of components, or processes, which are subject to a variety of failures. Thus, distributed algorithms for such system must be fault-tolerant. The design of such algorithms is a complex task, and their manual proofs of correctness are usually very complicated. Automatic verification of such systems is limited by the *state-explosion* problem, which becomes acute even for medium-size systems. Resent research in the area of *parametrized systems*, that is, systems that consist of potentially numerous instantiations of (usually) small and simple modules, uses techniques such as abstraction and deductive verification (see [9] for a survey). None of these techniques deal with fault-tolerance.

In this paper, we consider asynchronous distributed systems with undetectable crash failures of processes. We assume that there is a large number of processes, which can be placed into a small number of "equivalence classes," e.g., readers, writers, and servers (that store the value of a shared data object), so these systems can be regarded as parametrized systems. In a system with undetectable crash failures, a failed process stops communicating with the other processes, but the failure cannot be detected by the others. That is, a failed process is not distinguishable from one that is "very slow" [5]. We do not consider Byzantine failures, where a failed process can behave arbitrarily, and in particular, can deviate from its algorithm.

The distributed algorithms that we consider emulate shared registers and are based on quorum systems. In quorum systems, every broadcast operation awaits acknowledgments from only a quorum of processes, rather than from all processes. There exist different types of quorum systems. The common feature of all quorum systems is that an intersection of every two quorums is nonempty. Taking this feature as an axiom, we abstract away a particular quorum system. We propose a method for modeling and verification of quorum-based concurrent algorithms with crash failures by means of fully automatic model-checking procedures. Informally, our method is as follows. We express correctness (safety and liveness) properties of systems by quantified LTL formulae. We then show that the quantification over processes can be replaced by quantification over subsystems, each of which contains a constant number of processes and executes a constant number of requests, while maintaining semantic equivalence of the formulae. Model checking these small subsystems then suffices to conclude the correctness of the whole system. In general systems, we have to verify all possible subsystems, and since the number of subsystems is exponential in the number of processes, the verification cannot be done automatically for an arbitrary number of processes. The key idea, which allows us to check only a small number of subsystems, is that the systems we consider consist of a small number of "equivalence classes" of sim-

ilar processes. Two processes are placed in the same equivalence class if they run the same algorithm and are in the same state at the present time in the execution. Since server processes store the value of the register, the current value stored in a server is also taken into account when checking equivalence

Algorithms that we study in this paper use unbounded time-stamps. Thus, the straightforward implementation of each process yields a structure with an infinite number of states. However, since we only deal with subsystems with a constant number of processes, and quorum-based decisions are based solely on comparison between time-stamps (and not on their absolute values), we are able to abstract the absolute values of time-stamps away and to use only their relative values. Since the number of requests in subsystems is constant, the number of time-stamps is also constant, and thus we are able to express their comparative values by a constant number of boolean variables.

When we construct subsystems, we *project* the processes onto the subsystem, abstracting away all states that are unreachable and all variables that are unaccessible in this subsystem. The main problem in projecting the processes is that in a large system, processes perform transitions depending on the quorum-based decisions. Since in a constant-size subsystem a quorum of processes is inaccessible, we replace quorum-based decisions by non-deterministic decisions. This replacement creates illegal executions which do not exist in the original system. The first type of illegal executions is executions in which the client process non-deterministically moves to a state where the output is chosen before receiving the most updated reply. We filter such executions away by always including a process in the intersection of the current quorum with all other quorums in the small subsystem that we model-check. The second type is executions in which processes are stuck in the "waiting-for-replies" state and never terminate. We filter such executions away by replacing regular model-checking with fair model-checking, where the fairness constraint effectively expresses the condition "there is always a live quorum of processes".

An additional advantage of model-checking small subsystems is didactic. That is, the necessity of a part of the algorithm can be easily verified by removing this part from the model and model-checking the altered subsystem. If the altered subsystem still satisfies the desired property (non-vacuously), it means that the removed part was not necessary for the correctness of the algorithm.

2 Preliminaries

2.1 Temporal logic and Kripke structures

In model checking, we check whether a system given as a Kripke structure (labeled state-transition graph) satisfies a specification given as a temporal logic formula (or a finite automaton). In this paper we assume that the specifications are given in the linear temporal logic LTL [8,4]. The semantics of temporal logic is defined with respect to Kripke structures. We use Büchi fairness constraints, where an execution is *fair* if it visits a *fair* state an infinite number of times.

2.2 Registers

A *read/write* register (or simply, register) type supports an arbitrary set $Vals$ of values with an arbitrary initial state $v_o \in Vals$. Its invocations are $read$ and $write(v)$, for some $v \in Vals$. Its responses are v and ack. Its sequential specification f requires that every $write$ operation overwrites the last value written and returns ack (that is, $f(write(v), w) = (ack, v)$); and every $read$ operation returns the last value written (that is, $f(read, v) = (v, v)$). In a system consisting of processes $P_1, P_2, \ldots$, a process P_i interacts with a shared register by means of input actions of the form $read_i$ and $write_i(v)$ and output actions of the form v_i and ack. A read/write register is called k-reader/m-writer if only k processes are allowed to read, and m processes are allowed to write the register. We use the term multi-reader (multi-writer) when the number of readers (writers) is unrestricted. We now define several register properties. In our definitions, we talk about $read$ and $write$ operations. As opposed to I/O actions (which are atomic by definition), operations start when the request is placed in the system and terminate when the result is returned: the result of a $read$ operation is a value and the result of a $write$ operation is an indication that a $write$ has terminated successfully.

When expressing the register properties in quantified LTL, we denote by r_i the i-th read operation, by w_i the i-th write operation, and by s_i the i-th server. In order to express the order between processes, we introduce new boolean variables for beginning and end of each operation: $op_i.b$ ($op_i.e$) changes from **false** to **true** when the first (last) action of op_i is executed. We use Lamport's notation of arrows to express order between processes [7], where $op_1 \rightarrow op_2$ (op_1 strictly precedes op_2) is a shortcut for $\mathsf{G}(op_2.b \Rightarrow op_1.e)$, and $op_1 \leftrightarrow op_2$ (op_1 is concurrent with op_2) is a shortcut for $\neg(op_1 \rightarrow op_2) \wedge \neg(op_2 \rightarrow op_1)$, and $\Rightarrow$ denotes boolean implication. We also use variables $w_i.val$ for a value written by w_i, $r_i.val$ for a value read by r_i, and $s_i.val$ for a value that s_i holds. The values are pairs of the data value and a time-stamp, that is, $op_i.val =$

$\langle v, ts \rangle$. When comparing between values, we say that $op_i.val = op'_j.val$ iff $op_i.val.v = op'_j.val.v$ and $op_i.val.ts = op'_j.val.ts$. Let x be a register, and let σ be a sequence of invocations and responses of x. The following definitions of registers in a single-writer multi-reader system are taken from [7].

- **Safe register:** σ is *safe* if every complete *read* operation that does not overlap any *write* operation returns the latest written value or the initial value if no value has been written yet. A register x is *safe* iff all its traces σ are safe. Formally,

$$\forall w_i, r : \mathsf{G}[\ (w_i \rightarrow r) \wedge (\neg\exists w_j : w_i \rightarrow w_j \rightarrow r) \wedge (\neg\exists w_k : w_k \leftrightarrow r) \Rightarrow \mathsf{F}(r.val = w_i.val)\ ]. \quad (1)$$

  We note that we only express the condition of correct value returned when there are previous writes. When there are no previous writes, the requirement is expressed trivially.

- **Regular register:** σ is *regular* if it is safe and in addition every *read* operation that overlaps some *write* operations returns either one of the values written by overlapping *writes* or the latest non-overlapping value. A register x is *regular* iff all its traces σ are regular. Formally, this additional requirement is expressed as

$$\forall w_i, r : \mathsf{G}[\ (w_i \rightarrow r) \wedge (\neg\exists w_j : w_i \rightarrow w_j \rightarrow r) \Rightarrow \mathsf{F}(r.val = w_i.val) \ \vee\ \exists w_k : (w_k \leftrightarrow r \wedge \mathsf{F}(r.val = w_k.val))\ ]. \quad (2)$$

- **Atomic register:** σ is *atomic* if it is regular and in addition all invocations of σ are *linearizable* (see [6] for a definition) to a sequential register (that is, a register in which there are no overlapping operations). A register x is *atomic* iff all its traces σ are atomic. Formally, a register is atomic if it satisfies Equation 2 and in addition

$$\forall w_1, w_2, r_1, r_2 : \mathsf{G}[\ ((r_1 \rightarrow r_2) \wedge \mathsf{F}(w_2.val = r_1.val) \wedge \mathsf{F}(w_1.val = r_2.val)) \Rightarrow \neg\mathsf{F}(w_1 \rightarrow w_2)\ ]. \quad (3)$$

For the purposes of this work, we consider an operation to be *live* if the terminating state is eventually reached (that is, $\mathsf{F}op.e$ holds).

2.3 Characterization of algorithms

We start by characterizing the class of algorithms $\mathcal{R}$ to which our method applies. The characterization of $\mathcal{R}$ is as follows.

(i) Algorithms in $\mathcal{R}$ emulate a shared register by numerous instances of *server* processes.

(ii) There is a single writer process [3] .

(iii) Servers can crash and stop responding to requests. However, there is always a live *quorum* of servers.

(iv) Clients (writers and readers) are non-faulty.

(v) Each server can hold exactly one value at each point.

(vi) All values are accompanied by (unique) time-stamps.

(vii) Upon receiving different replies from servers, the output answer is chosen based on the time-stamps of the replies.

We do not restrict the number of rounds each client performs in order to read/write a value. We do assume, however, that this number is constant.

Each invocation of the algorithm (whether a read or a write) can be verified in isolation from the previous invocations. This observation allows us to model invocations by *micro-processes* [2]. A micro-process implements a single operation (e.g., a single read or write), after which it is destroyed. It has a very few states, and thus systems consisting of micro-processes can be easily model checked. The only problem with micro-processes is that a writer should store the current time-stamp. We circumvent it by assuming that the time-stamp is given to the writer micro-process as an input together with the input value. In single-writer algorithms, we can safely assume that this input time-stamp is unique and increases with each subsequent request. In multi-writer algorithms, more subtle reasoning is needed.

3 Automatic Verification of Quorum-Based Register Emulations

In this section, we present our main result, namely, that the desired safety and liveness properties can be automatically verified in small subsystems and the correctness of the whole system deduced from the correctness of these subsystems. This result follows from Lemmas 3.1 and 3.2.

Lemma 3.1 *Safety properties of algorithms from $\mathcal{R}$ can be verified by model checking of a finite number of subsystems which contain a constant number of readers and writers.*

Essentially, we prove that the temporal formulas that express the correctness properties of the register can be rewritten in the prenex way with universal quantifiers only. The pure LTL part of the formula involves at most

[3] Our work can be extended to the multi-writer case. For multiple writer processes, additional work should be performed in order to prove uniqueness of the time-stamp.

4 clients. The resulting pure LTL formulas are

$$\varphi_{safe} = \mathsf{G}[\ (w_i \rightarrow r) \wedge \neg(w_i \rightarrow w_j \rightarrow r) \wedge \neg(w_k \leftrightarrow r) \Rightarrow \mathsf{F}(r.val = w_i.val)\ ] \quad (4)$$

for safe register,

$$\varphi_{regular} = \mathsf{G}[\ (w_i \rightarrow r) \wedge \neg(w_i \rightarrow w_j \rightarrow r) \Rightarrow$$
$$\mathsf{F}(r.val = w_i.val) \vee (w_k \leftrightarrow r \wedge \mathsf{F}(r.val = w_k.val))\ ] \quad (5)$$

for regular register, and

$$\varphi_{at} = \mathsf{G}[\ ((r_1 \rightarrow r_2) \wedge \mathsf{F}(w_2.val = r_1.val) \wedge \mathsf{F}(w_1.val = r_2.val)) \Rightarrow \neg\mathsf{F}(w_1 \rightarrow w_2)\ ] \quad (6)$$

for atomic register. Since we model clients by micro-processes, each of which executes exactly one request, the number of requests in a subsystem is equal to the number of clients in this subsystem. We are not done yet, since it remains to prove that we are able to consider a constant number of servers as well.

Lemma 3.2 *The subsystems in Lemma 3.1 can be constructed with a constant number of servers. The number of servers participating in a single subsystem is bounded by 2^m, where m is the number of communication rounds between a client (reader or writer) and servers in this subsystem.*

The proof is based on the observation that at any given time during the execution there is a constant number of equivalence classes of servers. Since the decision is made based on the time-stamp and not the number of replies with the same value, we need only one representative from each equivalence class to be included in a subsystem. Since each communication round divides the set of servers by 2 (into servers that responded and servers that did not respond in this round), m rounds divide the set of servers by 2^m.

We note that Lemma 3.2 assumes that the current value is stored in a state of a server. This results in an unbounded number of states in the large system. In the constant-size subsystems that we consider, however, the number of values is is constant, and thus only a finite number of states of each server is reachable. We abstract away unreachable states. The number of rounds depends on the algorithms for readers and writers. For example, a reader micro-process can perform two communication rounds with the servers, where in the first one it reads the value and in the second one it writes it to the servers. We further note that the set of servers that did not reply to any of the clients can be ignored, as it does not play any role in the correctness of the system.

Combining Lemma 3.1 and Lemma 3.2 we are able to verify safety and liveness properties of algorithms by model-checking a constant number of sub-

systems with a constant number of (small) processes. It remains to show how quorum-based decisions are projected onto small subsystems. We do so by replacing quorum-based transitions by non-deterministic transitions. Clearly, such abstraction creates executions that are illegal in the global system. First, a process can now loop indefinitely in the state where it awaits replies from a quorum, thus creating a non-terminating execution. Second, it is now allowed to move to the state where it chooses the output value regardless of the number of replies it received.

We deal with the first problem by replacing the standard model checking procedure with fair model checking, where the fairness constraint allows only executions in which the state where a process awaits replies from a quorum occurs only a finite number of times. This is equivalent to the assumption that there is always a live quorum of servers. We deal with the second problem by a careful construction of the product Kripke structure. In our product Kripke structure, we want to eliminate all executions which are not consistent with receiving a quorum of replies. We rely on the definition of quorum systems, which postulates that every two quorums intersect. Thus, we construct the small subsystems so that for every pair of communication rounds in the subsystem, we include a single server process that lies in the intersection of the quorums of these two rounds. This ensures that the correct safety properties for this pair of rounds are verified, since they are enforced by this server.

Time-stamps Algorithms for register emulation usually use unbounded time-stamps. Fortunately, since our subsystems consists of a constant number of processes and a constant number of communication rounds is used, we are able to abstract away unbounded time-stamps. For correctness properties of safe, regular, and atomic registers in the single-writer systems it suffices to use 2 boolean variables for encoding the time-stamps. Indeed, the number of values written in the small subsystems does not exceed 4 for any of these properties.

Fine tuning Our previous reasoning allows us to reduce verification of a parametrized system to model-checking subsystems with a constant number of processes. We showed that it suffices to consider a subsystem with m communication rounds and 2^m servers. In fact, we can reduce the number of servers in the subsystem to $m(m-1)/2$ (the number of pairs of rounds). Indeed, the only servers that are essential for the proof of correctness of the algorithms in $\mathcal{R}$ are the ones in the intersections of quorums of a read and a write communication round.

4 Examples

In this section we illustrate our method on examples of algorithms that emulate a shared read/write register. We start with a simple example of a safe register in a single-writer multi-reader system. Then we extend it to the algorithm presented by Attiya et al. in [3].

The following codes of the client and server processes implement the safe register under the single writer assumption in a system in which there always exists a majority of live processes [7].

writer($\langle$**M**$\rangle$):
increase $TimeStamp$;
$count := 0$;
send $\langle M, TimeStamp\rangle$ to all;
upon receiving a reply ack do
$count := count + 1$;
until $count > n/2$,
where n is the number of processes;
return;

server($\langle$**M**$\rangle$):
upon receiving $\langle M\rangle$ do:
case $\langle M\rangle$:
$\langle M\rangle = ReadRequest$:
send $\langle M, TimeStamp\rangle$;
$\langle M\rangle = \langle m, TimeStamp\rangle$:
if $TimeStamp > LocalTimeStamp$,
then update M;
send ack;

reader():
$count := 0$;
send $\langle ReadRequest\rangle$ to all;
upon receiving a reply
$\langle M, TimeStamp\rangle$ do
$count := count + 1$;
save $\langle M, TimeStamp\rangle$;
until $count > n/2$,
where n is the number of processes;
choose the message M
with the maximal
time-stamp $MaxTimeStamp$;
return $\langle M, MaxTimeStamp\rangle$;

Recall that safe registers satisfy Equation 4, which is checked in subsystems of three writer processes and one reader process. Since each writer process invokes one write request and the reader process invokes one read request, there are 3 pairwise intersections of quorums of a write and a read requests, and thus 3 server processes in the subsystem. Moreover, there are three values written in this subsystem. Thus, the order between the three time-stamps can be expressed by three boolean predicates: $p_1 \equiv (w_i.val.ts > w_j.val.ts)$ and $p_2 \equiv (w_i.val.ts > w_k.val.ts)$, and $p_3 \equiv (w_j.val.ts > w_k.val.ts)$. Since the system has one (global) writer process, in the product subsystem the three write requests should strictly precede each other. Also, we do not have to consider the order in which $w_i \rightarrow w_j \rightarrow r$. The projection of a single write request (that is, a micro-process) and the projection of a single read request onto a subsystem with a constant number of processes are presented in the figures below. We assume that the correct time-stamp is given to the writer micro-process together with the input value (that is, val is a tuple $\langle v, ts\rangle$). The projection is obtained by abstracting away all variables that are unaccessible in the subsystem and computing the quotient abstraction of the resulting structure. The transitions $w_1 \rightarrow w_2$ and $r_1 \rightarrow r_2$ are non-deterministic. Recall, that we ensure liveness of the subsystem by replacing

regular model-checking with fair model-checking and safety by including the server that lies in the intersection of quorums in the subsystem. The server process stores the current value and its time-stamp in its state. Since in our subsystem there are three write requests, the projection of a server process onto a subsystem has a finite number of states. The projection of a server process on the subsystem with three write operations is presented in Figure 1.

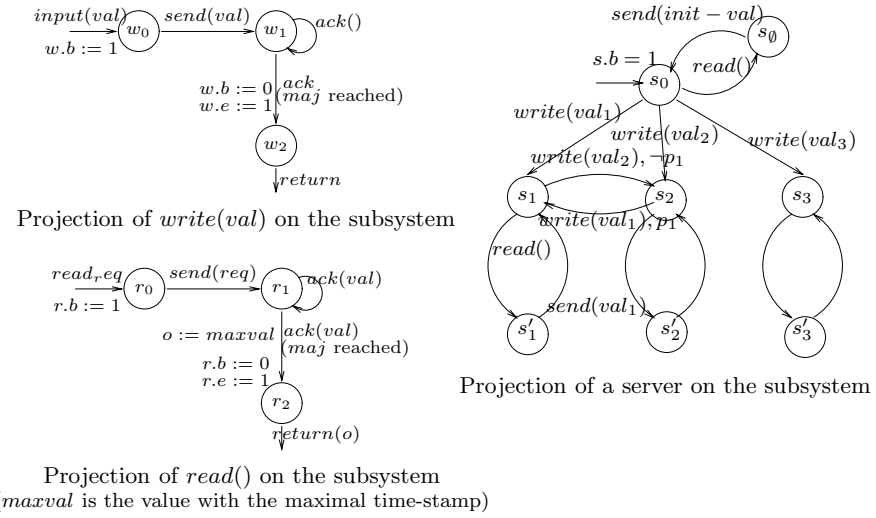
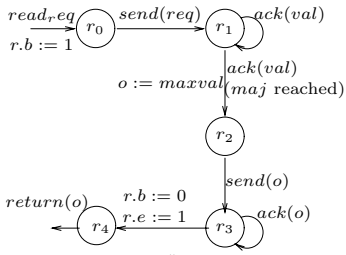


Fig. 1. Projection of processes in the safe register emulation

To avoid cluttering the figure, we omitted transitions similar to the ones between s_1 and s_2 and between s_1 and s'_1. All projections end in the *idle* state (omitted from the figure), in which they loop forever. The resulting subsystem is a product of 7 processes and has 3 write requests. The largest process is, thus, a server, whose size depends on the number of write requests. In this case, it has 8 states (2 states for each value, an initial state, and a state in which the initial value is returned). Thus, the straightforward implementation results in a subsystem that can be encoded with 21 variables. Filtering away interleavings that result in a vacuous satisfaction of the property and noting that replies from servers are assumed to be received in a known order results in a subsystem, which is the cross-product of a sequential composition of 3 clients and 2 servers (which has in total 25 states), one client (the writer w_k) and one server (which lies in the intersection of the quorums of w_k and r). Together with time-stamps, this subsystem can be encoded using 12 boolean variables, which is well within the reach of modern model-checkers.

The algorithm of Attiya et al. [3] for emulating a regular register in single-writer multi-reader asynchronous message-passing systems with crash failures

differs from the example we studied above in the code of the reader process. In this algorithm, the reader process performs two rounds: one read request and then one write request with the value received from the read request. The projection of the reader micro-process onto a subsystem is presented in the figure below. This difference leads to the atomicity of the register. Recall, that atomicity is expressed by Equations 5 and 6.



Projection of $read()$ on the subsystem ($maxval$ is the value with the maximal time-stamp)

Equation 5 is checked in a subsystem of three writer micro-processes and one reader micro-process. Note that the reader issues one read request and one write request, therefore the subsystem contains 4 server processes. The largest process in the subsystem is a server, which has 10 states, and thus the resulting subsystem can be encoded using at most 30 boolean variables.

Equation 6 is checked in a subsystem of two writer and two reader micro-processes. The subsystem contains 8 servers, thus the resulting subsystem can be encoded using at most 44 boolean variables. For both properties, applying the reasoning above further reduces the size of the subsystem.

The neccessity of the second round of read By abstracting away the second round of read and model-checking the resulting subsystem, we can show that in order to ensure the property of always returning the last preceding write value or a concurrent write value we only need one round of read. On the other hand, two rounds of read are essential for proving atomicity. Indeed, we can construct a subsystem with two writes combined sequentially and a read concurrent with the last write, in which Equation 5 holds, with automatically generated interesting witnesses to exhibit both types of satisfaction: there is an execution in which the read returns the value written by the first write, and another execution in which the read returns the value written by the second write. Since this is true for both reads regardless of their order, there exists an execution in which the first read returns the value of the second write and the second read returns the value of the first write.

5 Conclusions and Future Work

We proposed a method for modeling and verification of distributed algorithms for register emulation that allow crash failures of less than a quorum of servers. We argued that correctness (safety and liveness) properties of the whole system can be automatically model-checked in subsystems of constant size and then

extrapolated to the whole system. We avoided examining quorums of processes by replacing quorum-based transitions with non-deterministic transitions, and we showed how to filter away illegal executions that are created by using non-determinism. Modeling and automatically verifying distributed algorithms by means of small abstracted systems may help to determine what parts of the algorithm are really essential for its correctness by abstracting away a part in question and model-checking the resulting system. While the examples we considered in this paper are fairly simple, we believe that applying these methods to more complex algorithms may lead to interesting insights and even improvements of existing algorithms.

It remains to prove formally that correctness of small subsystems that we constructed implies the correctness of the whole system. The formal framework for these proofs is provided by the pairwise representation of concurrent systems [1]. The method of [1] can be generalized from analyzing products of pairs of processes to analyzing products of small numbers of processes.

References

[1] P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Trans. Program. Lang. Syst.*, 20(1):51–115, 1998.

[2] P.C. Attie. Synthesis of large dynamic concurrent programs from dynamic specifications. Technical report, Northeastern University, 2003.

[3] H. Attiya, Bar-Noy A., and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, Jan. 1995.

[4] E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, B:16, pp. 997–1072. MIT press, 1990.

[5] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[6] M. Herlihy and J.M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[7] L. Lamport. On interprocess communication – part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[8] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In *The Correctness Problem in Computer Science*, 215–273, 1981.

[9] L. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems. *Computer Languages - special issue* (to appear).