A NOVEL ABSTRACTION APPROACH TO MODEL CHECK DISTRIBUTED, FAULT-TOLERANT PROTOCOLS

Péter BOKOR
Advisor: András Pataricza

I. Introduction

We present an *abstraction* method to reduce the state space in *model checking* when verifying *fault-tolerant distributed protocols* meeting specified symmetry assumptions. The main idea of abstraction is that instead of storing the tuple of local states for each node of the system explicitly, we only store the sets (we call them *consistency sets*) of possible states a correct process can have at every round of communication. Representing sets instead of tuples leads to a considerable reduction of the state space, as shown in our experiments.

The abstraction is *sound*, i.e., if the model checker proves a property in the abstract model then it also holds in the non-abstracted one. Furthermore, for protocols where the latest communication (e.g., broadcast messages) between nodes fully determines the current system state (we call such protocols *memoryless*), soundness and *completeness* can be shown, i.e., a property holds if and only if the abstract property holds in the abstract system.

Due to information loss during the abstraction, we restrict ourselves to properties that require all processes to satisfy a certain property (we call such properties *uniform*). We make use of *temporal logic* to define properties that constrain the protocol's behavior in different stages (we call them *rounds*) of execution.

We also present results of experiments that were carried out with the SAL model checker.

II. Specifying Fault-Tolerant, Distributed Protocols

In this section we introduce the specification language (an adaptation from [1]) for the class of protocols we concentrate on, i.e., synchronous, frame-based and symmetric ones.

We consider fault-tolerant, distributed protocols containing n processors (or nodes). We assume that the number of faulty processors is a. The protocol proceeds through rounds, the number of rounds is finite, we denote it by r. At each round, every processor generates a message of the domain D and *broadcasts* to the others. We assume *synchronous* communication, i.e., at the round boundaries every message is supposed to be available. At every round, each processor maintains a local state of the domain S, which, at round 0, is one of the initial states ($S^I \subseteq S$). According to the current state, each node computes the message to send in the next round. This is done through the *deterministic* message generation function $m : S \rightarrow D$. Faults of faulty nodes manifest only through the messages they send. Faulty nodes do not obey the message generation function, they can send arbitrary messages, even *asymmetrically*.

Each processor computes its next state according to the state transition relation $\tau : S \times D^{n-a} \times S$. τ contains tuples of the *current* state of a correct processors, the messages received from other *correct* nodes and the *next* state of the processor. It is important to note that nodes also send messages to themselves. The messages of faulty nodes can affect the next states of the correct processors as follows. τ is a function in a fault-free environment: given the current state and the vector of messages, it contains exactly *one* tuple determining the next state of the node. In case faulty nodes are present, τ captures those state transitions that are affected by the messages sent by faulty nodes. Accordingly, nodes having the same input messages (from correct nodes), even with identical current states, can

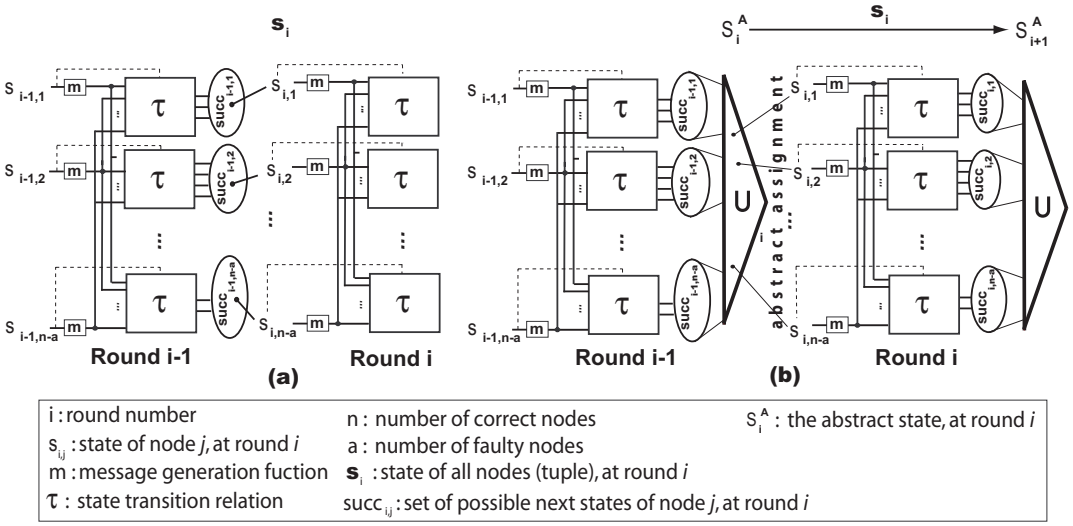

Figure 1: **(a)** Tuple-based mode **(b)**The abstraction with consistency sets

proceed to different states. By definition, every processor maintains the *same* τ. Based on the previous definitions we define a distributed protocol as a tuple: $DP = (S, S^I, D, n, a, r, m, \tau)$.

III. The abstraction principle

The non-abstracted model (we call it *concrete* model) is a straightforward derivation from the description of the protocol, which will turn out to be infeasible to verify properties against systems of larger size and/or in case of complex protocols. We make the following assumptions:

(A1) We assume a *pessimistic (Byzantine) fault model*, i.e., faulty nodes can send asymmetric information to correct nodes, and faults cannot be detected locally.
(A2) Correct nodes are allowed to start with *different initial states* (at round 0).
(A3) Faulty nodes are not modeled.

The main idea is that we model the state of each processor explicitly: a *tuple* of states will be updated at every round, where each element of the tuple stores the current state of the corresponding processor. Due to (A3) only correct nodes are modeled, accordingly, the size of the tuple is $n - a$. Figure 1(a) shows how the concrete model proceeds from round $i - 1$ to round i. The tuple $(s_{i,1}, ..., s_{i,n-a})$ is called the *(global) state* of the concrete model, where $s_{i,j}$ denotes the state of processor j at round i (which we call *local state* of processor j). Initially, every node starts from an initial state, which can be different for each node, in accordance with (A2). Every node broadcasts a message to the others, which is obtained by $m(s_{i,j})$.

If the current state $(s_{i-1,j})$ and the messages sent by correct nodes are available at the node's input, the set of possible next states will be computed: $succ_{i-1,j}$ is called the successor set of processor j, at round $i - 1$, containing all states that processor j can take at round i. These sets can vary from node to node, depending on the current state of the node and the messages sent by the faulty nodes, which are captured by τ. The global state at round i is $\mathbf{s}_i = (s_{i,1}, ..., s_{i,n-a})$, where $s_{i,j}$, the current state of processor j at round i, is taken from $succ_{i,j}$. Note that $s_{i,j}$ can be any of the possible states, since Byzantine nodes can send arbitrary messages, even asymmetrically (A1). Further development of the global state follows iteratively.

A. Consistency sets

In case of fault-tolerant, distributed protocols, the tuple-based model can easily blow the state space to exponential size, due to (1) the variety of possible *faulty behavior*s and (2) the case explosion

induced by the *permutation* of processors. To tackle case explosion we propose an abstraction, which *aggregates* the states that different nodes can have, instead of modeling the state of every correct processor explicitly.

The core idea is to use *consistency set*s, i.e., sets containing all states of *any* correct processors. More precisely, we define the abstract state to be a set containing all possible states that any correct processor can assume, given a certain communication pattern (i.e. the sequence of messages sent among the nodes). In other words, there is no run where two correct processes have two states owned by two different consistency sets. Such a set is considered *consistent* in the sense that it is an appropriate, common domain for all correct processors to take the current state from.

The idea of abstraction is depicted in Figure 1(b). Similarly to the tuple-based scheme, the successor set $succ_{i-1,j}$ is the set of all possible next states for processor j, at round i. In the concrete model the 1st processor proceeds to a particular state in $succ_{i-1,1}$, the 2nd processor to a state in $succ_{i-1,2}$, etc., and these states build a tuple constituting the global state of the next round. In the abstract model we aggregate these successor sets by building the *union* of them. We call this union (which is a set of states) consistency set, and we consider it as the *abstract state* (S_i^A). It is intuitive to see that such a set is consistent, since it contains all states that *any* correct node can assume at the given round.

To proceed from the abstract state S_i^A to S_{i+1}^A, the abstract model builds a tuple of states from S_i^A, which corresponds to a possible global state of the concrete system ($\mathbf{s}_i$). We call this global state $\mathbf{s}_i = (s_{i,1}, ..., s_{i,n-a})$ *abstract assignment*, where $s_{i,j}$ is chosen to be the next state of processor j. According to the states of $\mathbf{s}_i$ the next abstract state (S_{i+1}^A) is computed as the union of the successor sets $succ_{i,j}$. Note that the abstract assignment is an auxiliary construct that represents a possible assignment of states to correct processes. It is used to determine the set of messages produced at the next round. It is important to remark that the abstract assignment is chosen *non-deterministically*, since no assumption is made concerning what messages are sent by the faulty nodes.

Abstract assignments lead to unreachable global states, which is due to the fact that not every state in a consistency set is necessarily valid for a given processor. Such possibly unreachable states affect the completeness of the abstraction, which will be addressed later.

IV. The property space

Our basic model is that processors maintain program *variables* to run the protocol. Due to the assumption that nodes execute the same protocol, the set of variables are identical for every processor. Furthermore, we assume certain operators/relations over the domains of the variables to constrain their values. For example, if a state variable stores integer values, then we assume the definition of the ordinary operators $+, *$, etc. and relations $=, <, \leq$, etc. over integers.

We add a new interpretation to the notion of state. While the concrete model proceeds through the rounds of the protocol, the local state of the processors may differ at every round. Since at each node the state variables encode the distributed protocol DP, a state can be considered as an assignment of values to the variables. For example, one variable could be the program counter, which will be incremented in the next state at every node. A *property* is constructed upon state variables composed together via the pre-defined operators/relations.

In many practical cases, the desired properties define *uniform* requirements, i.e., it is required that *all* of the processors satisfy a certain property. For example, in case of a consensus protocol, agreement requires that every node adopts the same value after the protocol terminates.

Properties constrain the protocol's behavior at a particular round, i.e., in a particular state. Protocols often define requirements to constrain the behavior *over* rounds, i.e., for the sequence of states. For example, validity of consensus defines that if a node is correct then its proposed value (in the initial round) will be adopted by the others (in the final round). We make use of *Linear Temporal Logic* (LTL, originally introduced in [2]) to define such requirements.

The abstract counterpart of a property (we call it *abstract property*) is interpreted over consistency sets. We assume that the property is uniform and we define that an abstract property is true in an abstract state if all states in the consistency set satisfy the property.

V. Property preservation

An abstraction is *(strong) property preserving* if a property is true in the concrete model if and only if the abstract property holds in the abstract one. *Weak* preservation means the following: the fact that an abstract property holds in the abstract model implies that the corresponding property is true in the non-abstracted model, however, nothing is guaranteed if the abstract property is violated. It can be proven that our proposed abstraction is weak property preserving with respect to temporal uniform properties. The formal proof of this property is beyond the scope of this paper.

Furthermore, strong preservation can be proven if we restrict the class of distributed protocols. Generally, the next state of a processor depends on its *current* state and the *messages* received from the other processors. Memoryless protocols are protocols, where the next state is *not* dependent on the current state of the node.

VI. The main application: model checking

The main application of our abstraction is model checking. We assume a model checking environment that supports verification of LTL formulas. Instead of model checking if a property holds in the concrete model, we propose to build the abstract model and verify the abstract property. Initial experiments show that the abstract model reduces the size of the state space by several magnitudes also speeding up the model checking significantly.

The following table (Table 1) contains the results of model checking a 2-rounds, binary consensus protocol [3]. The protocol tolerates one malicious (Byzantine) sender if the number of nodes is greater than 3. The experiments were carried out with SAL's symbolic model checker [4].

nodes	Checking **agreement**				Checking **validity**			
	Concrete model		Abstract model		Concrete model		Abstract model	
	States	Time (s)	States	Time (s)	States	Time (s)	States	Time (s)
4	$1.34 \cdot 10^{08}$	13.48	1152	0.17	$1.34 \cdot 10^{09}$	12.92	4416	0.28
5	$3.43 \cdot 10^{11}$	23.87	5440	0.30	$5.84 \cdot 10^{12}$	21.55	$2.65 \cdot 10^{04}$	0.47

Table 1: Results of model checking a consensus protocols (the concrete and abstract models, resp.)

References

[1] N. A. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996.

[2] A. Pnueli, "The temporal semantics of concurrent programs," in *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, pp. 1–20, London, UK, 1979. Springer-Verlag.

[3] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, 27(2):228–234, 1980.

[4] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in *Computer-Aided Verification, CAV 2004*, R. Alur and D. Peled, Eds., vol. 3114 of *Lecture Notes in Computer Science*, pp. 496–500, Boston, MA, July 2004. Springer-Verlag.