Role-Based Symmetry Reduction of Fault-tolerant Distributed Protocols with Language Support*

Péter Bokor, Marco Serafini, Neeraj Suri, and Helmut Veith

Technische Universität Darmstadt, Germany
`{pbokor,marco,suri,veith}@cs.tu-darmstadt.de`

Abstract. Fault-tolerant (FT) distributed protocols (such as group membership, consensus, etc.) represent fundamental building blocks for many practical systems, e.g., the Google File System. Not only does one desire rigor in the protocol design but especially in its verification given the complexity and fallibility of manual proofs. The application of model checking (MC) for protocol verification is attractive with its full automation and rich property language. However, being an exhaustive exploration method, its scalable use is very much constrained by the overall number of different system states. We observe that, although FT distributed protocols usually display a very high degree of symmetry which stems from permuting different processes, MC efforts targeting their automated verification often disregard this symmetry. Therefore, we propose to leverage the framework of symmetry reduction and improve on existing applications of it by specifying so called role-based symmetries. Our secondary contribution is to define a high-level description language called FTDP to ease the symmetry aware specification of FT distributed protocols. FTDP supports synchronous as well as asynchronous protocols, a variety of fault types, and the specification of safety and liveness properties. Specifications written in FTDP can directly be analyzed by tools supporting symmetry reduction. We demonstrate the benefit of our approach using the example of well-known and complex distributed FT protocols, specifically Paxos and the Byzantine Generals.

1 Introduction

Model checking (MC) is a verification approach that exhaustively and automatically simulates the system by starting it from initial states and generating paths to verify that some specified properties hold along every path [7]. For designers of distributed systems, in particular of *fault-tolerant (FT) distributed protocols*, MC represents a useful tool for automatic verification of formal properties which are usually hand-proved. Not only can MC provide supporting evidence of the correctness of the proofs: it can also serve as a powerful tool for fast prototyping and debugging of protocols by showing counterexamples, i.e., runs violating

* Research supported in part by Microsoft Research, IBM Faculty Award and CASED.

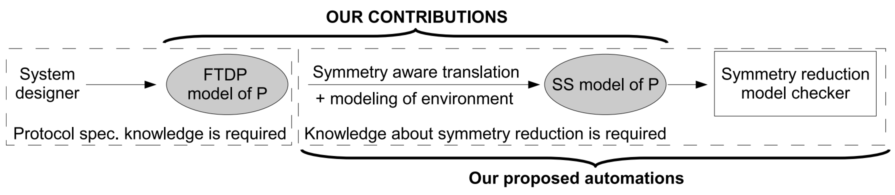

OUR CONTRIBUTIONS
System designer
FTDP model of P
Symmetry aware translation
+ modeling of environment
SS model of P
Symmetry reduction model checker
Protocol spec. knowledge is required
Knowledge about symmetry reduction is required
Our proposed automations

Fig. 1. The proposed approach for verifying a FT distributed protocol P

a certain property. However, the use of MC in the design and verification of distributed systems is still limited. In this paper, we identify (and tackle) two major barriers that prevent a widespread use of MC for distributed protocols.

The first barrier is that the design of a model still requires significant MC-specific expertise. Distributed systems designers typically use a pseudocode which primarily represents the process behavior and which abstracts many details required by the model checker. When the properties of an algorithm are hand-proved, only those predicates about the system and fault model which are needed in the proofs are enunciated. With automatic verification, however, the system and fault model need to be explicitly encoded into the model. This, together with the fact that the input languages of MC often require detailed understanding of the internal model representation used by the model checker, makes it difficult and error-prone for distributed systems designers to define models for MCs.

The second major limitation to the adoption of MC for verification of distributed systems is that the number of system states generated by the MC becomes very often unfeasible, i.e., *state space explosion*. Existing approaches to MC of FT distributed protocols disregard the fact that the *symmetric* nature of such systems can lead to significant state space reductions. For example, in many consensus protocols all processes execute the same algorithm and it is irrelevant to model which processes agree on a value as long as this is the same for all processes.

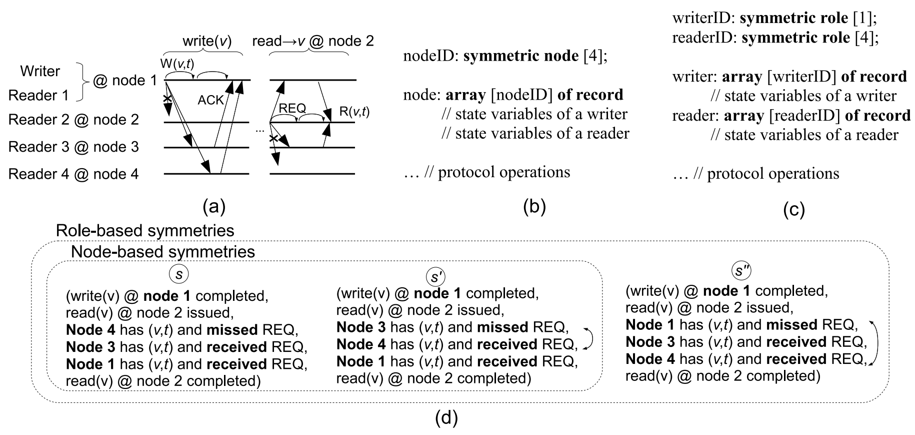
Paper Contributions This paper introduces the FTDP language which (a) allows writing models of FT distributed protocols using a simple pseudocode-like language and (b) forces the specification of symmetric systems to yield a sound and complete abstraction which effectively limits state space explosion. FTDP addresses both discussed barriers. It allows the system designer to concentrate on writing the pseudocode of the protocol behavior. The system and fault model can be defined by picking the desired properties from a palette of pre-defined templates which represent typical and well-known abstractions expressing the properties of the communication channels (e.g. synchrony, reliability) and of faults (e.g. crashes, Byzantine). FTDP also mitigates the state explosion problem by showing and discussing how *symmetry reduction* [10, 8, 16] can be integrated into automated model verification transparently to the system designer.

The basic idea of symmetry reduction is to identify groups of symmetric system states such that it is sufficient to explore one representative state in each group. However, the process of identification of symmetric states, commonly called *symmetry detection*, is a complex task. Automatic detection of symmetries in a model in order to produce a smaller, symmetry-reduced model is at least as complex as exploring the original model. It is therefore up to the designer to identify symmetries and express them in a language supporting symmetry reduction like SS [10]. FTDP identifies symmetries from the pseudocode of the system by leveraging *roles*. Roles are independent processes which partition the operations executed by the nodes participating in the protocol. They are explicitly defined, for example, in the Paxos protocol (leaders, acceptors and learners) [11] and in the OM protocol (commander and lieutenants) [15]. Implicit roles can be commonly identified in many distributed protocols (e.g., [1, 6, 18]). In our experimental evaluations we show that *role-based* symmetry reduction of distributed algorithms is very efficient as it can reach almost optimal state reduction for the detected symmetries. We also show that a role-based approach can be exponentially more efficient in the number of roles than the common, simplistic symmetry detection approach which considers nodes as the symmetric unit in the system [10, 8].

Our overall verification approach is depicted in Figure 1. The system designer writes a protocol's pseudocode using the FTDP language. By doing this, it also selects the appropriate system and fault model. FTDP is then automatically translated to the language SS. SS is general and comes with a precise proof of the correctness of symmetry detection. During the translation, FTDP uses roles to specify the symmetries of the system. The SS model of the protocol is then given as an input to a symmetry reduction model checker, which automatically explores the system state and verifies the properties.

In order to show the viability of the approach, we present the FTDP models of the Paxos and Oral Messages (OM) protocols. Both are fundamental consensus protocols which representatively show how FTDP can be used over different system models (asynchronous vs. synchronous) and over different fault models (crash vs. Byzantine). These protocols also demonstrate how roles are commonly used in distributed algorithms. Experimental verification shows that our approach can reduce the size of the state space of multiple orders of magnitude over non-symmetric models as well as over node-based symmetry reductions.

A Motivating Example We give the intuition of the proposed approach through the example of a simple reliable storage protocol. The protocol operations here, of different phases of information exchange and consequent decision/termination steps, are representative of a broad class of distributed FT protocols. This protocol implements a *regular read/write storage* (RS) assuming that only a strict minority of all processes can crash [17]. Channels are authenticated, i.e., a receiver process can identify the sender of the message. A message can be lost, duplicated or delayed but it cannot be forged. An RS is implemented on top of read/write registers each of them located on a different physical node. RS defines two roles, a single *writer* and *n readers* that can write/read to/from the RS respectively. Being a fault-tolerant solution, processes should be able to



```
nodeID: symmetric node [4];

node: array [nodeID] of record
        // state variables of a writer
        // state variables of a reader

… // protocol operations
```

```
writerID: symmetric role [1];
readerID: symmetric role [4];

writer: array [writerID] of record
        // state variables of a writer
reader: array [readerID] of record
        // state variables of a reader

… // protocol operations
```

s: (write(v) @ **node 1** completed, read(v) @ node 2 issued, **Node 4** has (*v*,*t*) and **missed** REQ, **Node 3** has (*v*,*t*) and **received** REQ, **Node 1** has (*v*,*t*) and **received** REQ, read(v) @ node 2 completed)

s': (write(v) @ **node 1** completed, read(v) @ node 2 issued, **Node 3** has (*v*,*t*) and **missed** REQ, **Node 4** has (*v*,*t*) and **received** REQ, **Node 1** has (*v*,*t*) and **received** REQ, read(v) @ node 2 completed)

s'': (write(v) @ **node 1** completed, read(v) @ node 2 issued, **Node 1** has (*v*,*t*) and **missed** REQ, **Node 3** has (*v*,*t*) and **received** REQ, **Node 4** has (*v*,*t*) and **received** REQ, read(v) @ node 2 completed)

Fig. 2. (a) Example of the regular read/write storage protocol. (b-c) Outline of the pseudocode of the same protocol using the usual node-based and the proposed role-based symmetries resp. (d) The benefit of the role-based approach: the node-based approach cannot detect that global state s'' is symmetric with global states s and s'.

access the RS even if some registers are unaccessible. Figure 2(a) sketches how the protocol works if $n = 4$ in a setting where the registers are located on the same physical nodes as the readers and where the writer and one reader are located on the same node. The writer can write a value v into RS by invoking write(v). This operation consists of requesting every register to update its value to v together with the writer's latest timestamp (t). The write completes if any majority of the registers have sent an acknowledgement. Even though reader 2 does not have v locally, it can use the RS protocol to obtain the value by invoking a read operation. The reader first requests every register to report the value with the latest timestamp, waits for a reply from a majority of all registers and returns the value with the largest timestamp among the replies. MC must verify that the RS is *regular*, i.e., a read always returns a value v that was actually written and v is not older than the value written by the last preceding write.[1]

Figure 2(b) shows the template of the pseudocode specification of the RS protocol used to detect node-based symmetries by construction. The designer specifies that the system consists of nodes, each of them able to host a writer and a reader. Nodes are declared to be symmetric, i.e., their local states are interchangeable. Symmetry violations are prevented in the specification language by disallowing the definition of symmetry-breaking operations. The global state of the system at each instant of time is given by an array storing the local state of each node of the system.

Figure 2(c), on the other hand, depicts the pseudocode-template of the same protocol with the ability of detecting role-based symmetries (like in FTDP). Every role, writer and reader, is defined to be symmetric. There is one writer

[1] An operation op_1 precedes op_2 if op_1 completes before op_2 is invoked.

process and four reader processes. The assignment of processes to physical nodes needs not be modeled because the properties of RS does not specify nodes. The global state of the system is the set of arrays, each of them belonging to a role, storing the local state of each role instance. The benefit of the role-based approach is demonstrated by an example in Figure 2(d). Three global states s, s' and s'' are shown which only differ regarding which node has missed a read request. *All* these global states are symmetric because each of them can be obtained from another by permuting the IDs of the readers. However, the node-based approach cannot detect that s'' is symmetric with s and s' because the model has to remember that reader 1, which is the only one hosted on the same node as the writer, has received the read request. Therefore, the model checker explores two states (s or s' and s'') in the node-based model. In contrast, it suffices to explore a single state (arbitrarily selected among s, s' and s'') in the role-based model.

Related Work A recent survey of general applications and tools for symmetry reduction is [16]. Our work is related to symmetry detection and to approaches that are specific to automated formal verification of FT distributed protocols. We assume that the model checker can distinguish between symmetric states; related techniques are also surveyed in [16].

The proposed solution assumes that the system consists of a finite number of processes. This strong assumption enables us to provide full automation and an expressive property language. Our recent brief announcement [4] provides a summary level overview of the approach. A powerful tool for the specification and the analysis of distributed systems is provided by the TLA+ language and the TLC model checker [13]. TLC supports symmetry reduction and requires that the user detects symmetries. FTDP models automatically detect symmetry and can be also translated into TLA+. +CAL [14] is a language allowing high-level specification of algorithms which, similarly to FTDP and SS, is automatically translated into TLA+. +CAL does not support the specification of symmetries and is lower-level (and also more general) than FTDP. For example, the modeling of message-based communication and faults must be implemented by the user.

Other work uses model checkers with no symmetry reduction support to verify consensus protocols under the crash fault model and the Heard-Of system model [23, 24]. The latter model assumes that a message which is sent in a communication round cannot arrive in later rounds. This additional assumption facilitates verification of consensus protocols, but is only sound for systems implementing it. The symmetry detection approach of FTDP can also be extended to exploit symmetry under the Heard-Of system model. Since FTDP restricts to finite models, the technique of abstracting protocols using infinite time stamps into a finite representation [23] can be combined with our approach.

Another work verifies the optimistic termination of Byzantine consensus protocols [25]. This approach differs from ours in many aspects: it is specific to consensus protocols, it uses a dedicated verification engine, it does not use symmetries, and it does not verify the entire protocol but rather focuses on opti-

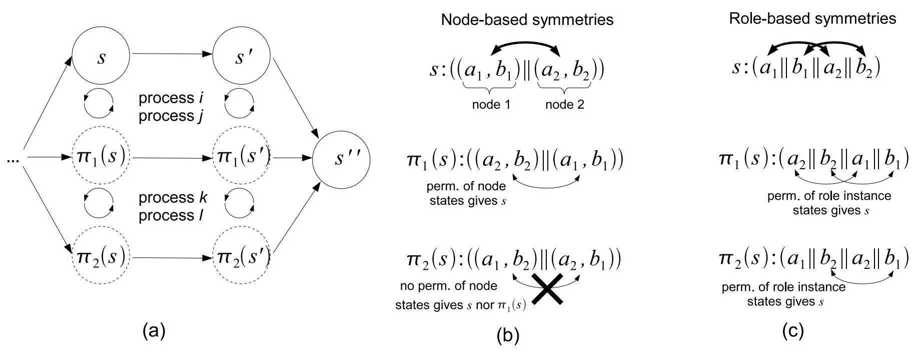

Node-based symmetries
Role-based symmetries
s
s'
s''
process i
process j
process k
process l
$\pi_1(s)$
$\pi_1(s')$
$\pi_2(s)$
$\pi_2(s')$
...
$s:((a_1,b_1)\|(a_2,b_2))$
node 1
node 2
$\pi_1(s):((a_2,b_2)\|(a_1,b_1))$
perm. of node states gives s
$\pi_2(s):((a_1,b_2)\|(a_2,b_1))$
no perm. of node states gives s nor $\pi_1(s)$
$s:(a_1\|b_1\|a_2\|b_2)$
$\pi_1(s):(a_2\|b_2\|a_1\|b_1)$
perm. of role instance states gives s
$\pi_2(s):(a_1\|b_2\|a_2\|b_1)$
perm. of role instance states gives s
(a)
(b)
(c)

Fig. 3. (a) Example symmetric state space with symmetry reduction. The dashed line states are not explored leading to state space reduction. (b) If the system has two nodes, each running instances of two roles a and b, where instances of a have states a_1 and a_2 and instances of b has states b_1 and b_2, symmetric state $\pi_2(s)$ is not detected in the classic node-based approach but (c) is detected in the role-based approach.

mistic cases. Model checking of self-stabilizing algorithms was proposed by using symbolic techniques that are (under favorable conditions) insensitive to the large number of initial states [22]. However, different but symmetric initial states need not be explored, which, if combined with explicit state model checking, does not suffer from the drawback of symbolic approaches. Work that uses MC to verify specific FT distributed protocols but that disregards symmetry (e.g., [21]) can naturally leverage our technique.

2 A Role-based Approach to Tackling Symmetry

Symmetries of Distributed Protocols A typical example of symmetries in the state space, or *state graph*, of the system is when all nodes in a distributed system execute the same process. The intuition is that a (global) system state s where two processes i and j assume different local states is symmetric with another state $\pi(s)$ where these two local states are swapped. Formally, *symmetry* [8] is a permutation π acting on all reachable system states satisfying that for every state s and its successor s' it holds that $\pi(s')$ is a successor of $\pi(s)$.[2] Figure 3 (a) shows a simple case of a symmetric state graph, where any pair among $s, \pi_1(s)$ and $\pi_2(s)$ is symmetric.

Symmetry reduction [16] exploits such symmetries to ease model checking. The idea is that the system exhibits indistinguishable behavior when started from symmetric states if the property under verification does not distinguish symmetric states. By using this technique a reduced model, or *reduced state graph*, is explored which is defined such that every state s of the reduced model

[2] More generally, the pairs of symmetric states s and $\pi(s)$ yield a bi-simulation for the state graph.

is a representative state among all states of the original model that are symmetric with s, and two states are connected in the reduced model if any two states of the corresponding sets of symmetric states are connected in the original model. In Figure 3(a) the reduced state graph contains only the representative states s, s', s'', and their relations.

Detecting Symmetries The definition of symmetry shows that all relations and states in the non-reduced state space may have to be visited to *automatically detect* symmetries, although this entails the very same complexity we want to eliminate. Therefore, we take the approach of creating symmetric models *by construction*, where the exploration of the state space is not needed because symmetries are indicated by the system designer. In order to indicate symmetries in the model the designer uses a special data type called *scalarsets*. Scalarsets were introduced in the SS language and define subranges with restricted operations, e.g., scalarset values can only be checked for equivalence and arrays with scalarset index type cannot be indexed by constants. The restrictions guarantee that any permutation of scalarset values results in symmetric states.

Efficient Symmetry Detection via Roles We observe that most FT distributed protocols are expressed, or can be easily expressed, in terms of *roles*. Protocols are executed by computing elements, termed processors or *nodes*. Each node executes one or more state machines, called *processes*, whose state consists of input and output message buffers and a local state. State transitions are atomic and are activated by either reading a message from the input buffer or by responding to an internal event (e.g. timeouts). Possible effects of a state transition are changing the local state of the process and writing new messages in its output buffer.

As protocols normally consist of a single process p_i per node $i \in [1, n]$, it is natural to model the system as the parallel composition $p_1 || \dots || p_n$ and to represent the current system state as a *configuration*, i.e., a tuple containing for each node i the local state of its process p_i. This is done by existing symmetry reduction approaches for distributed protocols, which use a single scalarset to represent the IDs of nodes in the current configuration (e.g. [10, 8]). Each node is thus modeled as an atomic entity and two configurations are symmetric if the states of the nodes can be permuted. We call this approach *node-based*.

The key idea of FTDP is that it lets the designer define a set R of *roles*, which are independent processes having non-intersecting states and whose state transitions are activated by non-intersecting sets of incoming messages and internal events. The behavior of each process p_i is expressed in FTDP as the parallel composition $p_i = r_i^1 || \dots || r_i^{k_i}$, where each r_i^j is an instance of a role in R and where each role has at most one instance per process. Our *role-based* symmetry detection identifies symmetries by permuting the states of multiple role instances rather than the states of nodes as a whole. FTDP models each role in R with a separated scalarset whose size is determined by the number of the corresponding role instances in the system. In other words, we do not model the local state of a node but the local state of each role instance separately. This is possible because the properties of the protocol specify roles rather than nodes.

We illustrate the difference between role- and node-based approaches using the example state graph of Figure 3 (a). All states s, $\pi_1(s)$ and $\pi_2(s)$ are symmetric but only a role-based approach may be able to detect all existing symmetries. Consider for example that these states model a protocol where each of the two nodes executes instances of two roles $\{a, b\} = R$ as in Figures 3 (b) and (c). A node-based approach can detect the symmetry of s and $\pi_1(s)$ because two node states are permuted, but no symmetry is detected between $\pi_2(s)$ and any of the previous states. This symmetry can be detected by our role-based approach because role instances, rather than nodes, are modeled as basic symmetric entities.

The role-based approach can yield an exponentially larger number of symmetric states compared to classic node-based approaches. In the best-case, the model reduced using the symmetries detected by the role-based approach contains less states than the original model by a factor of at most $\prod_{i=1\ldots|R|} n_i!$, where R is the set of roles and n_i is the number of processes executing role $r_i \in R$. This is because every state in the reduced model corresponds to at most $n_i!$ different states in the original model where n_i instances of r_i are permuted. In our experiments we show that this best-case reduction is almost reached for both Paxos and OM(1), so the identified reductions are very efficient. Furthermore, role-based detection can lead to an exponential best-case gain in terms of state reduction compared to the classic node-based approach in common systems. Assume that all n nodes execute all roles, i.e., $n_i = n$ for all i. In this case, the maximum benefit of symmetry reduction with the node-based symmetry detection approach is $n!$, which is $(n!)^{|R|-1}$ times less than with the role-based approach. Note that there is no guarantee that the reduced state space remains intractably large. In fact, symmetry reduction is able to mitigate state space explosion instead of fully tackling it.

3 The FTDP Language by Example: Modeling Paxos

Our goal is to create a language which, besides detecting symmetries, is able to faithfully model a broad class of FT distributed protocols. Models written in our language resemble the pseudocode of protocols so that system designers can easily use it.

We now present the FTDP language through the example of the Paxos protocol [11]. The complete FTDP model to verify the safety of Paxos is very compact and as depicted in Figures 4 and 5. The syntax of FTDP can be found in the Appendix.

Paxos solves the *consensus* problem, where each process keeps a local value and only one of these values is delivered to all processes. It assumes asynchronous, lossy channels with out-of-order delivery and at most a minority of processes which can crash. In a didactic paper [12] successive to [11], Lamport explicitly mentions three roles for each process, *leaders*, *acceptors* and *learners*. Leaders send a *proposal*, composed of the current local value and a proposal number,

Channel Models		
Synchrony	Asynchronous / Synchronous	(No / Existing) known upper bound on computation and message transmission delays
Reliability	Lossy / Reliable	Sent messages (are not / are) eventually received
Authentication	Authenticated	The receiver can identify the sender of the message
Delivery order	Out-of-order / FIFO	Sent messages (are not / are) received in the same order as they are sent
Duplication	No	Sent messages are not duplicated by the channel
Channel size	B-bounded	At most B messages can be sent but not yet received
Fault Models		
Status	Correct / Crash faulty / Byzantine faulty	Process always follows specification / Process eventually stops / Process does not follow specification

Table 1. Overview of system models available in the FTDP language

to all acceptors. An acceptor accepts a proposal only if it has not yet received any other proposal with a higher proposal number. A proposal is termed as *chosen* if a majority of acceptors accepts it. A chosen proposal can be learnt by the learners by collecting the accepted proposals from the acceptors. Consensus requires that (a) it is impossible that two proposals with different values are ever chosen (*safety*) and (b) a proposal is eventually learnt (*liveness*).[3]

Palette of System Models and API The FTDP language supports multiple common system models that usually appear in FT distributed protocols. We model systems as parallel compositions of processes, i.e., role instances, communicating via messages sent through point-to-point directed channels. A summary of the different channel and fault models that can be selected by the user in FTDP are depicted in Table 1. A global parameter of every FTDP model determines a specific channel model. A field called `status` denotes for each process whether the process is correct, crash faulty or Byzantine.

Incoming messages activating state transitions are referred by the special variable `msg`, which has a user defined message type. A set of variables of the form `2roleName[k]` is used for sending a message to the k^{th} process executing in the specified role (e.g., `2leader[k]`). These variables are written by the sender process and have a user defined message type.

FTDP Model Structure and Declarations Every FTDP model defines four blocks for the definitions of roles and message types (`type`), state variables (`var`), initial assignments of the variables (`init`), and rules updating the variables (`rules`). The first three blocks for Paxos are depicted in Figure 4. A role is defined for leaders and acceptors by the role name and the number of the corresponding role instances `m` and `n`, which are constant model parameters (line 1 and 2). In order to verify safety we only need to model that a proposal is chosen so learners are not modeled explicitly. The type of messages is defined between

[3] Note that in MC terms a liveness property differs from a safety property in that it can only be violated through infinite runs.

```
 1 type    leaderID: role[m];                                       // m processes are leaders
 2         acceptorID: role[n];                                     // n processes are acceptors
 3         Msg: record
 4                 msgType: enum {READ,READ_REPL,WRITE};            // message type
 5                 propNo: 1..m*L;
 6                 currPropNo: 0..m*L;
 7                 val: 0..m end;
 8 var     leader: array[leaderID] of record
 9                         propVal: 1..m;                           // proposed value
10                         propNoPool: array[1..L] of 1..m*L;       // proposal no. pool, L: size of the pool
11                         propNoID: 0..L;                          // index of current proposal no.
12                         readReplCnt:  0..⌈(n+1)/2⌉;              // READ-REPL message counter
13                         readReplHighestProp: record              // highest received prop. and its value
14                                 propNo: 0..m*L;
15                                 val:0..m end end;
16         acceptor: array[acceptorID] of record
17                         currPropNo: 0..m*L;                      // no. of the last accepted prop. , 0 is def. val.
18                         acceptedVal: 0..m;                       // val. of the last accepted prop. , 0 is def. val.
19                         highestPropNo: m*L end;                  // highest observed prop. no, 0 is def. val.
20 init
21 [](i_1:leaderID)[](i_2:leaderID)[](j_1:acceptorID)...[](j_⌈(n+1)/2⌉:acceptorID)  initRule:   // maj. of acceptors is corr (m=2,L=2)
22         if  i_1≠i_2 ∧ j_1≠...≠j_⌈(n+1)/2⌉  then                                        // acceptors are distinct
23                 undefine leader;   undefine acceptor;                                   // set all variables to undefine value
24                  leader[i_1].propVal:=1;   leader[i_2].propVal:=2;                     // leader i proposes i
25                  leader[i_1].propNoPool[1]:=1;   leader[i_1].propNoPool[2]:=3;         // init disjoint prop. no. pools
26                  leader[i_2].propNoPool[1]:=2;   leader[i_2].propNoPool[2]:=4;
27                  leader[i_1].propNoID:=0;   leader[i_2].propNoID:=0;
28                  leader[i_1].status=corr;   leader[i_2].status=corr;                   // leaders are correct
29                for  k:acceptorID  do
30                         acceptor[k].currPropNo:=0;
31                         acceptor[k].highestPropNo:=0;
32                        if  k=j_1∨...∨k=j_⌈(n+1)/2⌉
33                                then  acceptor[k].status:=corr  else  acceptor[k].status:=crash  endif endfor endif;
```

Fig. 4. Paxos modeled in FTDP — Declaration of types and variables, initialization with two leaders

lines 3-7. We define messages using a record where the value of the first field indicates the message type.

In every FTDP model, an array is defined for each role (lines 8-15 and 16-19), which stores the local states of role instances. The local state consists of the values of all state variables of the corresponding process. For example, proposal numbers are stored in an array called `propNoPool` (line 10). The FTDP model of Paxos is parametric in the size of this array (`L`). The Paxos protocol assumes that the sets of proposal numbers are disjoint for different leaders. We implement this by assigning in the `init` block distinct values in $[1..m \cdot L]$ to the elements of the `propNoPool` arrays.

We specify symmetry by declaring that the local states of role instances can be freely permuted. The syntax of the FTDP language guarantees that the permutation yields symmetric states. This results in state space reduction if two processes of the same role have different local states which can be permuted in two different (global) system states in the original model. For example, the local state of two acceptors can differ when only one of them receives a leader's message. On the other hand, role-based symmetry detection cannot achieve the theoretically maximum benefit when the two acceptors can have the same local state, for example because both receive the leader's message.

```
35 rules
36 [](i:leaderID) leaderElect:                                              // the protocol is initiated by leader
37         if  leader[i].propNoID<L  then
38                 leader[i].propNoID:=leader[i].propNoID+1;                 // take next proposal no.
39                 leader[i].readReplCnt:=0;                                 // reset reply counter
40                 leader[i]readReplHighestProp.propNo:=0;                   // reset the READ_REPL w/ highest prop. no.
41                 for  k:acceptorID  do                                     // leader initializes read phase...
42                         2acceptor[k].msgType:=READ;                       // ...by sending READ mess. to each acceptor
43                         2acceptor[k].propNo:=leader[i].propNoPool[leader[i].propNoID]  endfor endif;
44 []
45 [](i:acceptorID)[](j:leaderID) onReceiveREAD:                             // upon receipt of READ mess. at acceptor
46         if  msg.msgType=READ  then
47                 if  msg.propNo>acceptor[i].highestPropNo  then            // make a promise, otherwise discard mess.
48                 acceptor[i].highestPropNo:=msg.propNo;
49                 2leader[j].msgType:=READ_REPL;                            // acceptor i completes read phase...
50                 2leader[j].propNo:=msg.propNo;                            // … by sending a READ-REPL message to the leader
51                 2leader[j].currPropNo:=acceptor[i].currPropNo;
52                 if  acceptor[i].currPropNo>0  then  2leader[j].val:=acceptor[i].acceptedVal  endif endif endif;
53 [](i:leaderID)[](j:acceptorID) onReceiveREAD_REPL:                        // upon receipt of READ_REPL mess. at leader
54         if  msg.msgType=READ_REPL  then
55           if  leader[i].readReplCnt<⌈(n+1)/2⌉  then                       // leader waits for majority, else discards mess.
56                 leader[i].readReplCnt:=leader[i].readReplCnt+1;           // increment counter
57                 if  msg.currPropNo>leader[i].readReplHighestProp.propNo  then   // updates the highest prop. received so far
58                         leader[i].readReplHighestProp.propNo:=msg.currPropNo;
59                         leader[i].readReplHighestProp.val:=msg.val  endif;
60                 if  leader[i].content.readReplCnt=⌈(n+1)/2⌉  then        // if enough READ-REPL messages received
61                   for  k:acceptorID  do                                   // leader initiates the write phase...
62                         2acceptor[k].msgType:=WRITE;                      // ...by sending a WRITE mess. to each acc.
63                         2acceptor[k].propNo:=leader[i].propNoPool[leader[i].propNoID];
64                         if  leader[i].readReplHighestProp.propNo=0
65                                 then  2acceptor[k].val:=leader[i].propVal;
66                                 else  2acceptor[k].val:=leader[i].readReplHighestProp.val;  endif endfor endif endif endif;
67 [](i:acceptorID)[](j:leaderID) onReceiveWRITE:                            // upon receipt of WRITE mess. at acceptor
68         if  msg.msgType=WRITE  then
69           if  msg.propNo≥acceptor[i].highestPropNo  then                  // accept if no promise prohibits this, else discard mess.
70                 acceptor[i].currPropNo:=msg.propNo;   acceptor[i].acceptedVal:=msg.val  endif endif;

71 safety:  ∧_{val=1..2}  G(  ∃(i_1:acceptorID)...∃(i_⌈(n+1)/2⌉:acceptorID)
72                          i_1≠...≠i_⌈(n+1)/2⌉∧
73                          acceptor[i_1].currPropNo=...=acceptor[i_⌈(n+1)/2⌉].currPropNo∧
74                          acceptor[i_1].acceptedVal=val⇒
75                              (F  ∃(j_1:acceptorID)...∃(j_⌈(n+1)/2⌉:acceptorID)
76                                  j_1≠...≠j_⌈(n+1)/2⌉∧
77                                  acceptor[j_1].currPropNo=...=acceptor[j_⌈(n+1)/2⌉].currPropNo⇒
78                                          acceptor[j_1].acceptedVal=val   ))
```

Fig. 5. Paxos modeled in FTDP — Rules and safety property

State Initialization and Transitions Initialization rules are used to set the initial values of process variables and the fault model of each process. Since asynchronous communication and concurrent operations between leaders are more challenging to handle than leader crashes, the initialization rule of Figure 4 sets all leaders as correct and selects a minority of crash faulty acceptors (lines 21-33). For simplicity of the presentation, we assume that `m=2` and `L=2`. The initialization rule is parametric in the process IDs so that there is a distinct rule for each possible assignment of the parameters i_1, i_2 and $j_1 \ldots j_{\lceil (n+1)/2 \rceil}$ to process IDs. For example, in case of $n = 3$, the assignment $i_1 = 1, i_2 = 2, j_1 = 1, j_2 = 3$ determines one instance of the rule.

Each FTDP *rule* corresponds to a state transition which can update the local state and send messages. The rules for Paxos, separated by `[]`, are depicted in Figure 5. Rules are labeled for an easier reference. For example, the rule labeled as `leaderElect` (lines 36-43) handles an internal event triggered by leader election which makes leader i propose its value. Other rules are parameterized by i and j, the receiver and the sender of a message. Rules in FTDP can be guarded by a Boolean condition (lines 37, 46, 54, 68). For example, `leaderElect` is executed only if there is some unused proposal number left in the pool (line 37).

Temporal Properties in FTDP Safety in Paxos requires that once a proposal with a value *val* is chosen, no other proposal is chosen at some time in the future with a value different from *val*. Such properties can be naturally written in temporal logics. FTDP supports Computation Tree Logic (CTL*) which is a powerful temporal logic containing other useful logics like CTL or LTL. For example, safety can be defined in FTDP by using the temporal operators **G** ("always") and **F** ("eventually") (Figure 5, lines 71-78).[4] The basic assumption of role-based symmetries is that the property does not specify which role instance is executed by which physical node. In fact, the specification of such properties is impossible in FTDP as the model does not have the notion of nodes.

4 Symmetry Reduction of FTDP Models

The syntax of FTDP hides the modeling of channels and faulty processes from the user. These are modeled in the SS translation of FTDP models. In other words, FTDP defines syntactic sugar for SS. The translation between FTDP and SS guarantees that out-of-order delivery, message losses and process faults are considered in all possible ways. Therefore, no case can be overlooked which is necessary for a sound verification process. The soundness of verification is also affected by the property of the protocol. The property language of FTDP supports a broad class of properties that is provably preserved by symmetry reduction. We now give an overview of the translation between FTDP and SS and our property preservation results. The precise semantics of FTDP can be found in our technical report available online [3].

Faithful Model of Environment Channels are modeled via message buffers. An input buffer is an array or a multiset depending on whether the channel is FIFO or delivers messages out-of-order. The size of each input buffer is bounded by B. Output buffers correspond to the API variables in the form `2roleName[k]` and can contain a single message. The transmission of a message is modeled by moving it from the output buffer of the sender process into the input buffer of the recipient process. In case this buffer is full the message is discarded (lossy channels) or the sender must wait until all the messages it is sending can be copied into the input buffers of the recipients (reliable channels). We model authenticated channels by defining at each process a buffer for each other process. A process receives and processes a message by removing it from the input buffer.

[4] Note that because of the implication in line 77 it is not required that another proposal is ever chosen.

In lossy channels it is decided non-deterministically if a message in the input buffer is lost, in which case it is removed without processing.

A crash faulty process that has not yet crashed is modeled such that it correctly follows the process specification. Upon receiving a message from a crash faulty process, it is decided non-deterministically if the message is actually processed. If not, the sender is considered to be crashed and the message is discarded as it had not been sent. In such a way we also model scenarios where a process crashes after it has sent a message to only a subset of processes.

The state of a Byzantine process is not modeled. We model a process receiving a message from a Byzantine sender by non-deterministically selecting an arbitrary message from the domain. Thus, the size of this domain directly affects the size of the state space.

A system is synchronous if there is a known upper bound on message computation and delivery time, and is considered asynchronous otherwise. We model synchronous systems by assuming that a correct process waiting for a message is able to perfectly detect if the sender is faulty and the message will never arrive. Therefore, we introduce a Boolean flag `msg.absent` which is true if and only if the sender is either Byzantine faulty and fails to send a message or is crashed. The message itself (`msg`) contains the necessary information about which message is missing.

Property Preservation Symmetry reduction is sound to use only if it preserves the properties in FTDP, which is provided by the following theorem:

Theorem 1. *[10, 8] Let P be an SS model and AP a set of Boolean expressions defined over the variables in P. Given the state graph representation M of P, let the reduced state graph M_R be obtained from M by the permutation of scalarset values. M_R preserves every CTL* property f over AP, that is, f holds in M iff f holds in M_R.*

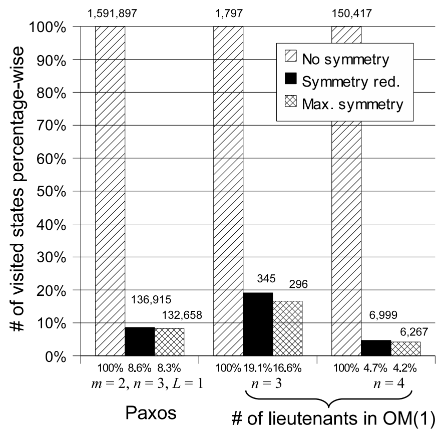
The property language of FTDP is essentially CTL* where the quantifiers, for example in the safety property of Paxos of Figure 5, are syntactic sugar for ANDs and ORs. Since the translation from FTDP to SS does not change AP, the above theorem justifies the soundness of our proposed approach as depicted in Figure 1. The details of the proof can be found in [3]. Note that every proposition p in AP is symmetric in that it cannot distinguish between specific role instances. Formally, in FTDP p must be quantified (existentially or universally) over the IDs of role instances. This constraint about AP enables the preservation of a class of properties as general as CTL*. We remark that the same constraint is needed if we restrict to a simpler class of properties such as simple invariants. Techniques where less symmetric properties can be preserved, at a price of less reduction, are surveyed in [16].

5 Experiments

We tested our approach on the representative synchronous and asynchronous Paxos and the OM(1) protocols. These two protocols represent a wide spectrum

Protocol	Param.	Property	Symm. red.	States	Gain	Effic.	Time	Result
Paxos	$m = 2$ $n = 3$ $L = 1$	safety	No	1,591,897			268 s	Verified
			Node-based	795,945	2x	33%	226 s	Verified
			Role-based	**136,915**	**12x**	**96%**	32 s	Verified
		Erroneous safety (chosen = accepted)	No	649,301			61 s	CE found
			Node-based	325,074	2x	-	226 s	CE found
			Role-based	**57,677**	**11x**	-	12 s	CE found
Faulty Paxos (always accept proposals)		safety	No	1,114,891			126 s	CE found
			Node-based	562,298	2x	-	122 s	CE found
			Role-based	**101,239**	**11x**	-	20 s	CE found
OM(1)	$n = 3$	IC1-2	No	1,797			0.1 s	Verified
			Node-based	941	2x	31%	3 s	Verified
			Role-based	**345**	**5x**	**85%**	0.1 s	Verified
	$n = 4$	IC1-2	No	150,417			9.6 s	Verified
			Node-based	26,401	6x	24%	17 s	Verified
			Role-based	**6,999**	**22x**	**90 %**	7.4 s	Verified
	$n = 5$	IC1-2	No	-			-	Out of mem.
			Node-based	2,402,167	-	-	4 h	Verified
			Role-based	**490,839**	-	-	2 h	Verified
Faulty OM(1) (two Byzantine faults)	$n = 3$	IC1	No	934			0.1 s	CE found
			Node-based	843	1.1x	-	2.9 s	CE found
			Role-based	**200**	**5x**	-	0.1 s	CE found

(a)



(b)

Fig. 6. (a) Results of model checking Paxos and OM(1) with Murφ using no, node-based and role-based symmetry reduction: "Verified" if the property can be proved, "Out of mem." if the state space explodes, and "CE found" if a counterexample was identified. (b) Comparison between the maximum and measured benefit of role-based symmetry reduction ($n = 5$, OM(1) see table).

of different system and fault models. Different from Paxos, OM(1) is a synchronous, Byzantine fault tolerant consensus protocol using reliable channels. It defines two roles, a single commander who proposes its local value and n lieutenants who will agree on the same value if at most one general or lieutenant is Byzantine and $n \geq 3$ (IC1). Moreover, if the commander is correct its local value must be the agreed value (IC2). Conditions IC1-2 are called the interactive consistency (IC) conditions. The full FTDP model of OM(1) can be found in the Appendix (Figure 8).

We used the Murφ symmetry reduction model checker [9] as it implements the SS language.[5] Since Murφ only supports invariants, i.e., properties that must hold in all states of the model, we instrumented our SS models by monitors to check properties containing temporal operators. Monitors save system states that are specified by the property (e.g., the first chosen proposal in Paxos) and use it as a reference in other states (e.g., where a new proposal is chosen). Properties that cannot be defined via invariants and monitors, like liveness in Paxos, cannot be verified using Murφ and thus are excluded from the experiments.

Murφ uses different heuristics to minimize the (time and space) overhead of checking whether a state is symmetric with a previously visited one. Therefore, Murφ might also expand states that are symmetric. The results of the experiments are depicted in Figure 6 using Murφ's "heuristic fast canonicalization" algorithm. The results of Figure 6(a) include a comparison of the node-based and role-based approaches, the verification of the properties of both protocols as well as false properties and fault-injected protocols where, for each case, a counterexample was found. Our experiments cover for both protocols those (non-trivial) settings that were feasible to verify with Murφ. All experiments were executed on DETERlab machines [2], equipped with a Xeon processor and 4 GB memory and running a Linux installation with Fedora 6 core. The results show that symmetry reduction was able to achieve a benefit of at least one magnitude in terms of the number of visited states. Furthermore, OM(1) with 5 lieutenants could not be verified without symmetry reduction because the queue of unexplored states ran out of memory.

Figure 6 also compares the size of the reduced model (in terms of the number of visited states) with the lower bound on the number of non-symmetric states, i.e., the theoretical maximum benefit of node-based and role-based symmetries (see Section 2). The proportion of these two numbers is called *efficiency* [10]. It can be seen that both protocols are almost optimally symmetric with respect to their roles (approaching 100% efficiency). This is also highlighted in Figure 6(b) where we compare the size of the reduced model (middle bar) with the lower bound (rightmost bar), and relate them to the size of the original model (leftmost bar). This comparison cannot be done for OM(1) with $n = 5$ because the size of the original model is unknown. We can observe that the difference between the achieved and maximum benefit is within 3% of the size of the original model.

The node-based models assume that the number of nodes equals $max\{n_i\}$, i.e., the maximum number of processes executing the same role. Nodes can be arbitrarily allocated for role instances as long as no node hosts more than one role instance of the same role. Our experiments show that the role-based symmetry detection approach is superior to the node-based one in terms of the number of explored states. Note that even in the case of OM(1), where the theoretical maximum benefit is the same for the node-based and role-based approaches, the measured benefit is considerably higher with role-based symmetry detection.

[5] Other symmetry reduction model checkers like SymmSpin [5] or SMC [20] can also be used if the SS translation of FTDP models is adapted for the input language of the model checker. The SMC model checker supports liveness properties as well.

Intuitively, this is because the node-based model has to remember whether a lieutenant is hosted on the same node as the commander.

6 Conclusion

We have created FTDP, a pseudocode-like specification language for FT distributed protocols which can be directly used to model check the target protocol against its properties without specific MC expertise. FTDP flexibly supports the most used system and fault models and is able to specify symmetries of the protocol if it is divided into roles, a term familiar to protocol designers. We have shown that FTDP can naturally and compactly specify complex and widely-used distributed protocols such as Paxos and OM. Our role-based symmetry detection approach can be exponentially more efficient than the node-based approach. Our experiments on the MC of these protocols have shown that they are highly symmetric with respect to their roles as the experienced benefit approaches the theoretical maximum, and that the reduction in terms of the number of visited states is very significant, that is, around one order of magnitude.

References

1. H. Attiya, A. Bar-Noy, D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.
2. T. Benzel et al. Design, deployment, and use of the deter testbed. In *DETER Community Workshop on Cyber Security Experimentation and Test*, 2007.
3. P. Bokor, M. Serafini, N. Suri, H. Veith. Role-based symmetry reduction of fault-tolerant distributed protocols with language support. TR-TUD-DEEDS-04-04-2009, http://www.deeds.informatik.tu-darmstadt.de/peter/FTDP_SR.pdf, 2009.
4. P. Bokor, M. Serafini, N. Suri, H. Veith. Brief announcement: Practical symmetry reduction of fault-tolerant distributed protocols. In *DISC*, 2009 (To appear).
5. D. Bonacki, D. Dams, L. Holenderski Symmetric SPIN. *Journal on Softw. Tools for Techn. Transfer*, 4(1):92–106, 2002.
6. M. Castro, B. Liskov. Practical Byz. fault tolerance. Proc. *OSDI*, pp. 173–186,1999.
7. E. Clarke, O. Grumberg, D. Peled. *Model Checking.* MIT Press, 2000.
8. E. M. Clarke, R. Enders, T. Filkorn, S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods Sys. Design*, 9(1-2):77–104, 1996.
9. D. L. Dill, A. J. Drexler, A. J. Hu, C. H. Yang. Protocol verification as a hardware design aid. Proc. *ICCD: Int. Conf. on Computer Design on VLSI in Computer & Processors*, pp. 522–525, 1992.
10. C. N. Ip, D. L. Dill. Better verification through symmetry. *Formal Methods Sys. Design*, 9(1-2):41–75, 1996.
11. L. Lamport. The part-time parliament. *ACM Trans. Comp. Sys.*, 16(2):133–169, 1998.
12. L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.
13. L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., 2002.
14. L. Lamport. Checking a Multithreaded Algorithm with +CAL. Proc. *DISC*, pp. 151–163, 2006.

15. L. Lamport, R. Shostak, M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
16. A. Miller, A. Donaldson, M. Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3):8, 2006.
17. G. Chockler, R. Guerraoui, I. Keidar, M. Vukolic, Reliable distributed storage. *Computer*, 42(4):60–67, 2009.
18. M. Serafini, N. Suri et al. A tunable add-on diagnostic protocol for time-triggered systems. Proc. *DSN*, pp. 164–174,2007.
19. A. P. Sistla, P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM Trans. Program. Lang. Syst.*, 26(4):702–734, 2004.
20. A. P. Sistla et al. SMC: A symmetry-based model checker for verification of safety and liveness properties. *ACM Trans. Softw. Eng. Methodol.*, 9(2):133–166, 2000.
21. W. Steiner et al. Model checking a fault-tolerant startup algorithm: from design exploration to exhaustive fault simulation. Proc. *DSN*, pp. 189–198, 2004.
22. T. Tsuchiya, S. Nagano, R. B. Paidi, and T. Kikuno. Symbolic Model Checking for Self-Stabilizing Algorithms. *IEEE Trans. Parallel Distrib. Syst.*, 12(1):81–95, 2001.
23. T. Tsuchiya, A. Schiper. Model checking of consensus algorithms. Proc. *SRDS*, pp. 137–148, 2007.
24. T. Tsuchiya, A. Schiper. Using BMC to verify consensus algorithms. Proc. *DISC*, pp. 466–480, 2008.
25. P. Zielinski. Automatic verification and discovery of byzantine consensus protocols. Proc *DSN*, pp. 72–81, 2007.

A The FTDP Language

A.1 The Syntax

Figure 7 depicts the BNF syntax of FTDP. As usual, the operator "[]" is used to select array elements and "." to address fields of a record. The names of non-terminals specific to FT distributed protocols are prefixed by "`FTDP`". The following types are pre-defined in every model: the roles used by the protocol (⟨`FTDP-roleDecls`⟩) and the type of a message (`Msg`). The first type defines for each role ⟨`FTDP-roleType`⟩ the number of role instances. A message is modeled through a record of fields. For simplicity, the language allows the definition of one type of message only. This is not a limitation because the same type can be used to model various messages. Every FTDP model maintains for each role an array ⟨`FTDP-roleState`⟩ to store the local state of each role instance.

Every FTDP model must define at least one initial state. This is done via ⟨`rule`⟩. A simple rule is defined by a sequence of statements (⟨`stmt`⟩) and labeled for easier reference. A statement is used to update the values of process variables. Rules are separated by using the [] operator. Parameterized rules can be defined by writing []((⟨`id`⟩ : ⟨`typeExpr`⟩). This means that a rule is defined for every possible value of `id`. The execution of a rule means that the statements defined by the rule are executed. The state of the protocol is updated through the execution of a rule. If multiple rules are defined any of them can be executed. This is how the FTDP language supports non-determinism.

```
<FTDP>                  ::= type <FTDP-roleDecls>
                                Msg: record <decls> end;
                                <decls>
                            var <FTDP-roleState>
                            init <rule>
                            rules <FTDP-rule>
<FTDP-roleDecls>        ::= <FTDP-roleType> : role [<num>]; {<FTDP-roleType> : role [<num>];}
<FTDP-roleType>         ::= <id>
<FTDP-roleState>        ::= <FTDP-roleName>: array[<FTDP-roleType>] of  record <decls> end
                          | <FTDP-roleState> ; <FTDP-roleState>
<FTDP-roleName>         ::= <id>
<FTDP-rule>             ::= <FTDP-onReceipt>
                          | <FTDP-onTransition>
                          | ELSE
                          | [] (<id>:<typeExpr>) <FTDP-rule>
                          | <FTDP-rule> [] <FTDP-rule>
<FTDP-label>            ::= string
<FTDP-onReceipt>        ::= [](i:<FTDP-roleId>)[](j:<FTDP-roleId>) <FTDP-label> :
                          if <FTDP-guard> then <stmt> endif
<FTDP-guard>            ::= <boolExpr>
<FTDP-onTransition>     ::= [](i:<FTDP-roleId>) <FTDP-label> :  if <FTDP-guard> then <stmt> endif
<decls>                 ::= <id>: <typeExpr> {; <id>: <typeExpr>}
<typeExpr>              ::= <id>
                          | <num> .. <num>
                          | bool
                          | record <decls> end
                          | array [<num> .. <num>] of <typeExpr> | array [<id>] of <typeExpr>
                          | enum {<id>{, <id>}}
<rule>                  ::= <FTDP-label> : <stmt>
                          | [] (<id>:<typeExpr>) <rule>
                          | <rule> [] <rule>
<stmt>                  ::= <var> := <term>
                          | undefine(<var>)
                          | if <boolExpr> then <stmt> {else <stmt>} endif
                          | for <id>:<typeExpr> do <stmt> endfor
                          | <stmt> ; <stmt>
<boolExpr>              ::= <term> = <term>
                          | <term> > <term>
                          |   ¬<boolExpr>
                          |   <boolExpr>∧<boolExpr>
                          |   ∀ (<id>:<typeExpr>) <boolExpr>
                          |   ∃(<id>:<typeExpr>)<boolExpr>
                          | isundefined(<boolExpr>)
<term>                  ::= <var> | <num>
                          | <term> + <term> | <term> * <term>
<var>                   ::= <id> | <var>.<id> | <var> [<term>]
<id>                    ::= string
<num>                   ::= string
```

Fig. 7. The syntax of the FTDP language

Name	Type	Scope	Usage	Description
`msg`	`Msg`	`rules`	R	Latest mess. received by proc. i from j
`msg.absent`	`bool`	`rules`	R	True iff the mess. was not sent
`roleName[`i`].status`	`{corr,crash,byz}`	`init`	RW	Status of proc. i in role "roleName"
`2roleName`j	`array[roleId`j`] of Msg`	`rules`	W	Proc. i's mess. to "roleName$j$" instances

Table 2. List of predefined variables in FTDP modeling a protocol with k roles ($j \in [1..k]$). Variables can be read-only (R), read/write (RW), and write-only (W). Proc. i is defined in the FTDP rule.

The statement `undefine(v)` can be used to assign a special undefined value to all values in variable `v`. The predicate `isundefined(v)` is used to check if all values in `v` assume the undefined value. Otherwise, a variable `var` can be assigned a value `val` by writing `var:=val`. Conditional and iterative statements are defined similar to ordinary programming languages.

Rules that describe how correct processes or processes that have not yet crashed update their variables are defined via ⟨`FTDP-rule`⟩. These rules can be combined similarly to ⟨`rule`⟩. The rule ⟨`FTDP-onTransition`⟩ defines an event handler of the i^{th} role instance in the specified role (⟨`FTDP-roleId`⟩) to update its local state in response to an internal event. Another rule ⟨`FTDP-onReceipt`⟩ defines an event handler for the receipt of a message. The rule is parameterized by a reference to the receiver and sender role instances (i and j). Both rules are guarded by a Boolean condition (⟨`FTDP-guard`⟩) to govern which handler is executed. In order to avoid deadlocks in our models the `ELSE` rule can be used which executes the empty statement if no other rule can be executed.

The list of pre-defined variables and their description is depicted in Table 2. The use of these variables is better explained in Section 3 and in [3].

Symmetry by Construction We define conditions C1-C5 to ensure that the symmetry of an FTDP model is not broken. These conditions can be verified through simple syntactic checking. We assume that they are (automatically) checked by an interpreter.

(C1) An array with role index type can only be indexed by a variable of exactly the same type.
(C2) A term of role type may not appear as an operand to + or any other operator in a term.
(C3) Variables of role type may only be compared using =.
(C4) For all assignments $d := t$, the types of d and t may be (possible different) subranges or exactly matching roles.
(C5) Variables, elements of arrays and fields of records written by any iteration of a "for" statement indexed by a role type must be disjoint from the set of variables, elements of arrays and fields of records referenced (read or written) by other iterations.

The Property Language We adopt the Computation Tree Logic (CTL*) [7] to define properties of FT distributed protocols. Properties of an FTDP model can be defined via the CTL* temporal operators based on AP (atomic properties), where AP contains all Boolean expressions in the FTDP language. We refer to our technical report [3] for more details about CTL*.

There are two types of CTL* formulas, state formulas (which are true in a specific state of the system) and path formulas (which are true along a path of consecutive system states). The syntax of state formulas is summarized here:

- If $p \in AP$, then p is a state formula.
- If f and g are state formulas then $\neg f$, $f \wedge g$ and $f \vee g$ are state formulas.
- If f is a path formula, then $\mathbf{E}f$ ("there is a path") and $\mathbf{A}f$ ("for all paths") are state formulas.

Path formulas can be defined via the following two rules:

- If f is a state formula, then f is also a path formula.
- If f and g are path formulas, then $\neg f$, $f \wedge g$, $f \vee g$, $\mathbf{X}\ f$ ("next"), $\mathbf{F}\ f$ ("eventually"), $\mathbf{G}\ f$ ("always"), $f\ \mathbf{U}\ g$ ("until") and $f\ \mathbf{R}\ g$ ("release") are path formulas.

A.2 The Semantics

The semantics of SS is given via state graphs [10]. We show a translation which translates an FTDP model into an SS one. In this way every FTDP model determines a state graph. The semantics of FTDP properties is defined following the standard semantics of CTL* [7].

Preliminary into SS The syntax of the SS language [10] is similar to FTDP without the protocol specific details. In SS the user can also define *aliases*, i.e., short names of long variable references. In addition, a special data type called *scalarset* is available in SS. A scalarset is a subrange of natural numbers with restricted operations. The restrictions are essentially C1-C5 with scalarsets rather than roles. C1-C5 ensure that the behavior of the program is invariant under arbitrary permutation of the elements of a scalarset. When a new scalarset is declared, its size n is specified (n must be finite), and it represents a subrange from 1 to n with restricted operations. Another data type in SS is *multiset.* Multiset is an abstraction of an unordered array. The operations of the array, i.e., read and write access, are restricted in order to satisfy that the array is unordered. In fact, a multiset variable can only be accessed through the following special operators: `choose`, `multisetAdd`, and `multisetRemove`. (`choose id:mset`) is used to non-deterministically select the identifier `id` of an element in a multiset `mset`. The element can be referred by writing `mset[id]`. The functions `multisetAdd(msetElem,mset)` and `multisetRemove(id,mset)` are used to add a new element `msetElem` and to remove a previously selected one, respectively. Another function `multisetCount(mset,pred)` can be used to count the number of elements that satisfy the Boolean predicate `pred` in `mset`.

Translation From FTDP to SS The translation between FTDP and SS models is shown in Table 3. First, the field `absent` (line 2) is defined in the message record type to implement the detection of send omissions. The pre-defined variable `status` is implemented as a variable with enumeration type in the local state of each role instance; the constant `crashed` is defined to distinguish between processes that can potentially crash or have crashed already.

We define at each role instance a buffer for incoming messages from all other role instances (lines 7-9). For simplicity, Table 3 depicts the translation when channels can deliver messages out-of-order. Therefore, multisets are used to represent input message buffers. Assuming B-bounded channels, the syntax `multiset[B]` is used to declare a multiset with size B. The implementation of FIFO channels is discussed in [3]. Output buffers are defined as auxiliary variables (line 13, 24 and 42). In SS, the user can define auxiliary variables, introduced after the `with` keyword, that are defined in the scope of a rule and whose values are not included in the state of the system.

Rules are executed only if the process is correct or not yet crashed (lines 12, 22 and 40). Upon handling an internal event, the statements defined in the event handler of the FTDP model are simply copied into the SS model (line 15). This is followed by a routine that copies each message that is sent by the i^{th} role instance into the incoming buffer of the recipient process (line 16 and lines 47-53). Note that this is only done if the recipient process is not crashed nor Byzantine nor is the buffer full.

The translation is more complicated if the event handler receives and processes an incoming message. A distinction is made whether or not the sender of the message is Byzantine. Two rules, `label1` and `label2`, are generated in SS to handle these cases. In the first rule (`label1`) where the sender is not Byzantine (lines 18-37) the received message is non-deterministically chosen from the incoming buffer corresponding to the sender of the message (line 20). The message is renamed to `msg` for compliance with the FTDP model (line 21). The rule is parameterized with two additional Boolean flags, `senderCrash` (line 18) and `msgLoss` (line 19) to decide whether the sender has crashed and whether the message is lost, respectively. The flag `msgLoss` is only defined if lossy channels are modeled. In this case, only messages that are not lost are processed (line 25); otherwise the message is simply removed from the buffer. For simplicity, Figure 3 depicts the translation when channels are lossy. Details about the translation with reliable channels can be found in [3].

If `senderCrash` is true, then the status of the sender of a message is set to `crashed` (line 30). Only processes that are marked as crash faulty are allowed to crash. It is possible that a process had crashed after sending a message to a subset of processes. This is also decided by the `senderCrash` flag. If the flag is set the statements defined by the event handler are not executed nor messages are sent unless the system is synchronous and the absence of the message can be detected (lines 28-32). Either way the message is removed from the buffer (line 36). In the second rule (`label2`) where the sender is Byzantine (lines 39-46) message `msg` is a parameter of the rule (line 39). This is because the message buffers only contain messages sent by non-Byzantine processes. A Byzantine sender is modeled in a way that a rule is generated for every possible message in the domain. The translation simply copies the statements and includes the routine to send messages (lines 44-45).

Finally, the translation substitutes every occurrence of the FTDP keyword `role` with `scalarset`. This is how SS is able to detect role-based symmetries. Since C1-C5 are equivalent with the scalarset restrictions, our translation produces valid SS models.

B Property Preservation of Symmetry Reduction in FTDP Models

Let $M = (S, R)$ and L be the state graph representation of an FTDP model (according to Section A.2). The reduced state graph $M_R = (S_R, R_R)$ (also called "quotient graph") is obtained from M based on the symmetries induced by the scalarset data types [10], where every $[s] \in S_R$ is an equivalence class containing all states symmetric with s and $([s], [q]) \in R_R$ iff $((s', q') \in R)$ for some $s' \in [s]$ and $q' \in [q]$. L_R is defined such that, for every $[s] \in S_R$, $L_R([s]) = L(rep([s]))$, where $rep([s])$ is the representative state in $[s]$. The semantics of a CTL* formula f in M_R and L_R follows the standard semantics of CTL* [7]. Our property preservation result can be stated as follows:

Theorem 1 *If M, L and M_R, L_R are the corresponding state graphs and labeling functions of an FTDP model and f is an FTDP property, then $M \models f$ iff $M_R \models f$.*

We use two Lemmas to prove Theorem 1. Formally, a symmetry π in M is a permutation acting on S. A symmetry π is invariant for atomic property $p \in AP$ if $p \in L(s)$ iff $p \in L(\pi(s))$ for all $s \in S$. Our first Lemma appears as "Theorem 4.1" in [8]:

Lemma 1. *If the atomic properties are symmetric, i.e., every symmetry in $M = (S, R)$ that is used to obtain M_R is invariant for every atomic property p occurring in a CTL* formula f, then $M \models f$ iff $M_R \models f$.*

To prove Theorem 1, we have to prove that every atomic property p in an FTDP property f is symmetric. Let f_p be an invariant property, i.e., $f_p = \mathbf{G}(p)$. f_p in SS is defined via an additional rule that navigates the system into a special state called *error* if p is false in the current state of the system. Therefore, f_p and M implies the following state graph: $M^p = (S \cup \{error\}, R^p)$ such that $R^p = R \cup \{(s, error) | p \notin L(s)\}$. Our second Lemma appears as "Theorem 3" in [10] and it proves the following:

Lemma 2. *Every π which is induced by the permutation of scalarset values is a symmetry in M^p.*

Lemma 2 implies that $(s, error) \in R^p$ iff $(\pi(s), error) \in R^p$. Therefore, applying Lemma 2 to every p in f and using the definition of R^p imply the precondition of Lemma 1.

BLOCK	FTDP CODE	SS CODE

type	*Msg*: **record** decls **end**;	*Msg*: **record** 1
		        absent: Boolean; *2
		        decls 3
		    end; 4

var	roleName1: **array**[*roleId1*] **of record** decls **end**;	roleName1: **array**[*roleId1*] **of record** 5
		        status: **enum** {corr,crash,crashed,byz}; 6
		        *msgBuffer_roleName*1: **array** [*roleId1*] **of multiset**[*B*] **of** *Msg* 7
		        ... 8
		        msgBuffer_roleNamek: **array** [*roleIdk*] **of multiset**[*B*] **of** *Msg* 9
		        decls **end**; 10

rules	[](*i*:*roleIdR*) label:
	if guard **then** stmt **endif**
		[](*i*:*roleIdR*) label: 11
		if *roleNameR*[*i*]. *status*≠*byz*∧*roleNameR*[*i*]. *status*≠crashed∧guard **then** 12
		with *2roleName1*: **array**[*roleId1*] **of** *Msg*; … *2roleNamek:* **array**[*roleIdk*] **of** *Msg*; 13
		        undefine *2roleName1*; … **undefine** *2roleNamek*; 14
		        stmt; 15
		        MACRO_SEND; 16
		endif 17

rules	[](*i*:*roleIdR*)[](*j*:*roleIdS*)
	label: **if** guard **then** stmt **endif**
		[](*i*:*roleIdR*)[](*j*:*roleIdS*)[](*senderCrash*: *Boolean*) 18
		[](*msgLoss*: *Boolean*) label1: 19
		choose *ind*: *roleNameR*[*i*].*msgBuffer_roleNameS*[*j*] **do** 20
		alias *msg*: *roleNameR*[*i*].*msgBuffer_roleNameS*[*j*][*ind*] **do** 21
		if *roleNameR*[*i*]. *status*≠byz∧*roleNameR*[*i*]. *status*≠crashed∧ 22
		      roleNameS[*j*]. *status*≠byz∧guard **then** 23
		with *2roleName1*: **array**[*roleId1*] **of** *Msg*; … *2roleNamek:* **array**[*roleIdk*] **of** *Msg*; 24
		  if !*msgLoss* **then** 25
		        undefine *2roleName1*; … **undefine** *2roleNamek*; 26
		        msg.*absent*:=false; *27
		        if *senderCrash*∧*roleNameS*[*j*]. *status*≠corr **then** 28
		                msg.*absent*:=true; *29
		                if *roleNameS*[*j*].*status*=crash **then** *roleNameS*[*j*].*status*:=crashed **endif**; 30
		                stmt; *31
		                MACRO_SEND; *32
		        else stmt; 33
		                MACRO_SEND; **endif**; 34
		  endif; 35
		  multisetRemove(*ind*,*roleNameR*[*i*].*msgBufferS*[*j*]) 36
		endif endalias endchoose 37
		[] 38
		[](*i*:*roleIdR*)[](*j*:*roleIdS*)[](*msg*: *Msg*) label2: 39
		if *roleNameR*[*i*]. *status*≠byz∧*roleNameR*[*i*]. *status*≠crashed∧ 40
		      roleNameS[*j*]. *status*=byz∧guard **then** 41
		with *2roleName1*: **array**[*roleId1*] **of** *Msg*; … *2roleNamek:* **array**[*roleIdk*] **of** *Msg*; 42
		        undefine *2roleName1*; … **undefine** *2roleNamek*; 43
		        stmt; 44
		        MACRO_SEND; 45
		endif 46

MACRO_SEND	**for** *l*:*roleId1* **do if** ¬**isundefined**(*2roleName1*[*l*])∧*roleName1*[*l*]. *status*≠crashed∧*roleName1*[*l*]. *status*≠byz∧ 47
	        multisetCount(*roleName1*[*l*].*msgBuffer_roleNameR*[*i*],*true*)<*B* **then** 48
	                multisetAdd(*2roleName1*[*l*],*roleName1*[*l*].*msgBuffer_roleNameR*[*i*]) **endif endfor**; 49
	... 50
	for *l*:*roleIdk* **do if** ¬**isundefined**(*2roleNamek*[*l*])∧*roleNamek*[*l*]. *status*≠crashed∧*roleNamek*[*l*]. *status*≠byz∧ 51
	        multisetCount(*roleNamek*[*l*].*msgBuffer_roleNameR*[*i*],*true*)<*B* **then** 52
	                multisetAdd(*2roleNamek*[*l*],*roleNamek*[*l*].*msgBuffer_roleNameR*[*i*]) **endif endfor**; 53

Table 3. The SS translation of an FTDP model of a protocol with k roles assuming out-of-order and lossy channels. Lines marked with “*” are only used for synchronous systems.

C The FTDP Model of Byzantine Generals

The *Oral Messages (OM)* protocol solves consensus in a synchronous environment with reliable channels and Byzantine processes (called Byzantine Generals) [15]. The protocol is parametric in a, and we write OM(a), which is the number of Byzantine processes in the system. OM defines two roles, a single commander and n lieutenant processes. Any process can be Byzantine but their number is bounded by a. Briefly, the OM(a) protocol works like this: the commander in OM sends an order to all lieutenants who exchange the value of this order in subsequent communication rounds. The number of communication rounds is given by a. Figure 8 shows the FTDP model of the OM(1) protocol.

Consensus is defined through two properties under the condition that $n \geq 3a$ (see Figure 8). The first property of OM (called IC1 — "IC" comes from Interactive Consistency) defines that all lieutenants obey the same order. We define this by requiring that eventually every pair of correct processes decides for an order and this order must be the same. The second property of OM (called IC2) defines that all lieutenants obey the order sent by the commander if the commander is correct.

```
type     commanderID: scalarset[1];                                      // one commander
         lieutenantID: scalarset[n]                                      // n lieutenants
         Msg: record comm: Boolean end
var      commander: array[commanderID] of record
                   command: Boolean end;                                 // commander's order
                   hasProposed: Boolean;                                 // flags that corr. commander has sent order
         lieutenant: array[lieutenantID] of record
                   commands: array[lieutenantID] of Boolean;             // commands received by lieutenant
                   decision: Boolean; end                                // decision: accepted order
init
[](i:lieutenantID)[](comm:Boolean) init1:                                // corr. commander orders "comm"
         undefine lieutenant;
          for k:commanderID do
            commander[k].command:=comm;
            commander[k].hasProposed:=false;
            commander[k].status:=corr;
         endfor;
          for k:lieutenantID do                                          // i^th lieutenant is Byzantine
                   if  k=i  then  lieutenant[k].status:=byz  else  lieutenant[k].status:=corr  endif endfor;
[]
[](i:commanderID) init2:                                                 // Byzantine commander
         undefine commander;
         undefine lieutenant;
            commander[i].status:=byz;
         for k:lieutenantID do  lieutenant[k].status:=corr  endfor;      // correct lieutenants
rules
[](i:commanderID) propose:                                               // correct commander sends order
  if  ¬commander[i].hasProposed  then
         for k:lieutenantID do  2lieutenant[k].comm:=command  endfor;
         commander[i].hasProposed:=true
  endif
[]
[](i:lieutenantID)[](j:commanderID) recProp:                             // correct lieut. i receives order from commander
  if  isundefined(lieutenant[i].commands[i])  then                       // if lieutenant has not already received it
    if  msg.absent                                                       // order stored at commands[i] at lieut. i
         then  lieutenant[i].commands[i]:=defVal                         // defVal upon send omission
         else  lieutenant[i].commands[i]:=msg.comm                       // store received order otherwise
    endif;
    for k:lieutenantID do                                                // forward order to other lieutenants
         if  k≠i  then  2lieutenant[k].comm:=lieutenant[i].commands[i]  endif endfor
  endif
[]
[](i:lieutenantID)[](j:lieutenantID) recComm:                            // correct lieut. i receives command from lieut. j
  if  isundefined(lieutenant[i].commands[j])  then                       // if i has not already received it
    if  msg.absent
         then  lieutenant[i].commands[j]:=defVal
         else  lieutenant[i].commands[j]:=msg.comm
    endif;
    if  ∀(k:lieutenantID)¬isundefined(lieutenant[i].commands[k])  then          // majority voting
         if  $\exists(i_1:lieutenantID)\ldots\exists(i_{\lceil(n+1)/2\rceil}:lieutenantID)\, i_1\neq\ldots\neq i_{\lceil(n+1)/2\rceil}\wedge$
             $lieutenant[i].commands[i_1]=\neg defVal\wedge\ldots\wedge lieutenant[i].commands[i_{\lceil(n+1)/2\rceil}]=\neg defVal$
                   then  lieutenant[i].decision:=¬defVal
                   else  lieutenant[i].decision:=defVal                         // defVal if no majority exists
    endif
  endif;
[]
ELSE

IC1: F(  ∀(i:lieutenantID)∀(j:lieutenantID)                                     // agreement is always achieved
           lieutenant[i].status=corr ∧ lieutenant[j].status=corr ⇒
           ¬isundefined(lieutenant[i].decision) ∧ ¬isundefined(lieutenant[j].decision) ∧
           lieutenant[i].decision=lieutenant[j].decision   )
IC2: G(  ∀(i:commanderID)∀(j:lieutenantID)                                      // correct comm.'s order is decided
           commander[i].status=corr ∧ ¬isundefined(lieutenant[j].decision) ⇒
           commander[i].command=lieutenant[j].decision   )
```

Fig. 8. OM(1) and its properties specified in FTDP