Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 251 (2009) 49–64
www.elsevier.com/locate/entcs

Formal Model–Driven Design of Distributed Algorithms

Morten Kühnrich[1]

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
9220 Aalborg
Denmark

Abstract

We examine the use of formal model–driven development for creation and improvement of distributed algorithms. We use the integrated modeling and verification tool UPPAAL as our supporting tool. Instead of the traditional design, code and test-cycles known from agile paradigms, we employ formal model, verification and correction-cycles. The success of this approach is demonstrated on a distributed agreement algorithm from 1996 by Chandra and Toueg. We improve the number of communication rounds needed in the best-case from n to 2 where n is the number of agents. We end the paper with a correctness argument for systems with n agents. Formal model–driven development thus seems to be a fruitful approach for development of distributed algorithms.

Keywords: Model Driven Development, Fault Tolerance, Distributed Consensus, Distributed Agreement, Model Checking, Formal Verification.

Introduction

Agile software development follows an iterative pattern of design, implementation, test and correction. For engineering of distributed algorithms we examine a variant of the agile paradigm with formal model support (see Figure 1). It is related to model–driven development (MDD) [10] with the following differences: i) our view is on developing algorithms while MDD focuses on software development, ii) we do not use model transformations from models to executable code—we are concerned with correctness of the model.

Since our focus is on development of distributed algorithms with formal model support, we first create a formal model of the initial design idea. By model we mean a system with a clear semantics, in this paper a finite state system. Afterwards we verify the model against a correctness specification. In this paper verification is

[1] Email: mailto:mokyhn@cs.aau.dk

doi:10.1016/j.entcs.2009.08.027

performed by model checking. This is not a crucial choice, other techniques may be applied instead. We iterate until a design/model with correct behavior is found. From the model we extract the final design D expressed in high level pseudo–code. The correctness proof then refers to D.

We illustrate this cycle on an example from distributed computing. Distributed agreement is the following problem: a fixed number n of agents each propose a value v_i, $1 \leq i \leq n$. At some time everyone must agree on one common value v_i. We require *Termination*, i.e. every live agent eventually picks a value and *Agreement*, i.e. all live agents decide on the same value and *Validity*, i.e. the final value selected was suggested by one of the agents ealier on. Various solutions to this problem play an important role in modern computing. An example of that is Lamport's algorithm Paxos which is used in Google's Chubby distributed lock service [4].

In previous work [8] we studied a process-algebraic proof methodology by Francalanza and Hennessey [6] and used an agreement algorithm by Chandra and Toueg [5] as the underlying case study. The aim of that paper was to evaluate different proof strategies on a concrete distributed agreement algorithm, i.e. different ways of proving already known theorems. In this paper we extend and improve the algorithm by iterative development. In the end we provide a correctness proof for the extension.

We need three iterations in the cycle of Figure 1 and devote one section per iteration in the text below. The first iteration is based on Chandra and Toueg's initial **design**. We **model** the system[2] as a finite state automaton in UPPAAL

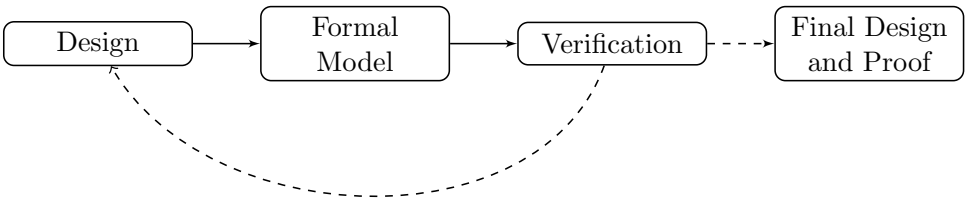

Fig. 1. Phases of tool supported development.

[2] (an integrated tool for specification, simulation and verification of timed automata) and **verify** the *Termination*, *Agreement* and *Validity* property using linear temporal logic. Our second iteration is intended to improve the design, but it turns out that the "improvement" is erroneous. The model checker generates a counterexample which helps us to i) understand the error and ii) fix it. In our third and last iteration we correct the model and design accordingly. Both termination and agreement is satisfied and the best case running time is improved. From there we find a general proof of correctness. This paper illustrates how formal model–driven development is of promising use in the domain of distributed algorithms.

[2] An archive of the UPPAAL code can be found at `http://www.cs.aau.dk/~mokyhn`.

Related work

In [9] Lynch the development of distributed algorithms is introduced using communicating automata. Proofs are handcrafted and verification is not applied.

In [12] Tsuchiya et. al. use the model checker Spin [3] to verify asynchronous agreement algorithms with a bounded but unknown number of rounds. Verification of agreement and termination is reduced to model checking of single phases of the given algorithm. Our choice of the tool UPPAAL opens perspectives for future work in the sense that qualitative properties involving time may be studied. An overview paper on agile tool support for modeling, simulation, verification and prototyping of distributed programs is found in Hasselbring's article [7]. Petri nets are used in connection with the verification part. As opposed to the sometimes cumbersome codings of data structures in Petri nets we find the simplicity and richness of UPPAAL data structures more convenient.

1 First Iteration

We now introduce the algorithm from [5]. Assume that each agent has a unique identity number p, $1 \leq p \leq n$ and a proposed value v_i.

We use communicating finite state automata with asynchronous fair message passing as the basic model. Fairness ensures that agents cannot wait forever to receive messages already sent. All n agents communicates over a medium where messages are not lost or altered under transmission.

Agents are however vulnerable to crashes which stops them for ever. Each agent has access to an *unreliable failure detector* which may help to detect whether other agents have crashed. We know this phenomenon from reality — pinging a server and getting a timeout does not necessarily mean that the server is down. The failure detector has the weak accuracy property: at least one live (i.e. never crashing) agent is never wrongly suspected (other live agents may be wrongly suspected). We call this agent the *trusted immortal* (in short TI) [11] which is a non-deterministically chosen agent from the set $\{1, \ldots, n\}$. When agent p uses the failure detector to suspect agent q we write **suspect** q. The suspicion command blocks if p suspects itself or it suspects TI. Otherwise **suspect** q acts as `skip`.

1.1 Design

Define Π as the set $\{1, \ldots, n\}$ of agents. We use $\perp$ for unknown values and define a set $\mathcal{V}_\perp$ as $\{1, \ldots, n, \perp\}$ and let $\leq_{nat}$ be the ordering relation on the natural numbers. The order $\leq \subseteq \mathcal{V}_\perp \times \mathcal{V}_\perp$ is the least relation containing $\leq_{nat}$ with the additional requirements $\perp \leq \perp$ and $\perp \leq i$ for $i = 1, \ldots, n$. An n-vector is a map from the set $\{1, \ldots, n\}$ to the set $\mathcal{V}_\perp$ where $\overline{\perp}$ is the n-vector $(\perp, \ldots, \perp)$. The order $\leq$ is extended point-wise to n-dimensional vectors and we write $V_1 \leq V_2$ when vector V_2 is greater than or equal to V_1. Uppercase letters such as V or Δ are used for vectors.

Chandra and Toueg	UPPAAL	Role
n	`n`	Number of agents
V	`V[n]`	Knowledge
Δ	`D[n]`	Relay
r	`r`	Round
p	`p`	Who am I
q	`q`	Iterator
decide	`decide`	Decision value

Table 1
Variables of the algorithm

1.1.1 Variables of the algorithm

See Table 1 (we use the UPPAAL part later on). A vector V holds the current knowledge of agent p (the *knowledge vector*). If $V(i) = v$ then agent p knows that agent i proposed value v. Vector Δ is used to relay knowledge from last round (the *relay vector*). Each agent has a round variable r which allows agents to order messages.

Every agent knows its own value, i.e. $V(p) = v_p$ and Δ initially equals V. The agreement algorithm presented in Table 1 has three phases of communication and computation.

Phase 1 — obtaining knowledge. First agents broadcast and update their knowledge during $n-1$ rounds. When a received message contains a previously unknown value then both knowledge and relay vector (line 10-12) are updated. Learned values are only relayed once because of the boolean predicate in line 10 and the fact that the relay vector is reset in the start of each round. It can be proved that every agent p that completes Phase 1 at least has the same knowledge as TI.

Phase 2 — correcting knowledge. If $V_i(j) = \perp$ for some agent i and j then either agent i has suspected agent j using the failure detector or j has crashed before sending messages to i. Such "not-known" values are distributed among all the participants. An agent k which receives knowledge vector V_i corrects coordinate j to $\perp$, i.e. $V_k(j) = \perp$. Destruction of knowledge in this fashion happens in line 22. It can be proved that every agent p which reaches the end of Phase 2 has the same $\perp$'s as TI. As an effect it holds that $V_{\mathrm{TI}} = V_p$ for any such p at the end of Phase 2.

Phase 3 — selecting the final value. The two phases above ensure that the knowledge vector of every live agent will be equal to V_{TI} at the time when they have completed Phase 2. The first non-zero value in the knowledge vector is chosen. Since the

trusted immortal knows it's own value i.e. $V_{\mathrm{TI}}(\mathrm{TI}) = v_{\mathrm{TI}}$ this value cannot be $\perp$. Hence every live agent will agree on a number in the end.

Definition 1.1 Define the following correctness properties:

(i) *Termination:* Every live process eventually decides some value.

(ii) *Agreement:* All live agents decide on the same value.

(iii) *Validity:* If a process decides value v, then v was proposed by some process.

Theorem 1.2 ([5]) *Algorithm 1 satisfies the termination, agreement and validity property. When run on n agents it terminates after n rounds in worst and best case.*

Algorithm 1 Distributed Agreement [5].

Require: An agent number $p \in \{1, \ldots, n\}$
Ensure: A final decision value in the interval $1, \ldots, n$

$V_p \leftarrow \overline{\perp}$, $V_p(p) \leftarrow v_p$, $\Delta_p \leftarrow V_p$

Phase 1:
for all $r_p \leftarrow 1$ to $n-1$ **do**
  send message $\mathsf{Phase1}(p, r_p, \Delta_p)$ to everyone
  $\Delta_p \leftarrow \overline{\perp}$
  for all $1 \leq q \leq n$
    block until
      receive $m = \mathsf{Phase1}(q, r, \Delta)$
      if $V_p(q) = \perp \wedge \Delta(q) \neq \perp$ **then**
        $V_p(q) \leftarrow \Delta(q)$
        $\Delta_p(q) \leftarrow \Delta(q)$
    or suspect q

Phase 2:
send message $\mathsf{Phase2}(V_p)$ to everyone
for all $1 \leq q \leq n$ **do**
  block until
    receive $m = \mathsf{Phase2}(V)$
    if $V(q) = \perp$ **then** $V_p(q) \leftarrow \perp$
  or suspect q

Phase 3: decide $= \min\{q \mid V_p(q) \neq \perp\}$

Remark 1.3 There can only be at most two values occuring at the same coordinate of two different knowledge or relay vectors. That is $\perp$ or an unique integer value. This is clear since a value on say coordinate i is only transferred to the same coordinate in other vectors. If we inspect the code in Algorithm 1 we observe that the actual values of the initial proposals v_i is unused. The only property used is

whether a given coordinate contains a $\perp$ or a value (in line 10 and line 20). From these two obervations we conclude that it is safe to let the initially proposed values for agent p equal p, i.e. $v_p = p$ for all $1 \leq p \leq n$. This assumption greatly reduces the size of the statespace in the model checking problem studied later on.

1.2 Formal Model.

Figure 2 represents Algorithm 1 as a finite state automaton. The UPPAAL model (see [2] for a tutorial on UPPAAL) contains n parallel instances of this automaton, one for each agent.

Each transition in each automaton is equipped with a *guard* (written with a normal font) and an *action* (written in bold) which updates the store. For now think of every object marked with dashed lines as if they were removed. Table 1 has a reference of the UPPAAL variables used.

Initially the automaton is in its `Idle` location. It initializes by a transition to location `Phase1`. From there it may either crash or move to location `Wait1` waiting for a message or suspecting. It loops through all agents, moves up a round, and returns to location `Phase1`. When all rounds are complete it moves to `Phase2`. At location `Wait2` all agents are looped through again. When finished it moves to `Phase3` selecting the final value.

Crashes are coded using location `Crash` which can occur from the locations `Phase1`, `Phase2`, `Wait1` and `Wait2`. Each crash transition is equipped with a local integer variable `c` which holds a non-deterministically selected value 0 or 1 (written `c:int[0,1]`). The TI never crashes, but for every other agent, if `c` is 1 then a crash is permitted (and otherwise not). The reader might wonder why variable c is needed. The reason is this: if a deadlock occurs from a location where the crash transition is the only one enabled, then UPPAAL forces the agent to crash. But this is not intended, since possible deadlocks should not force agents to crash.

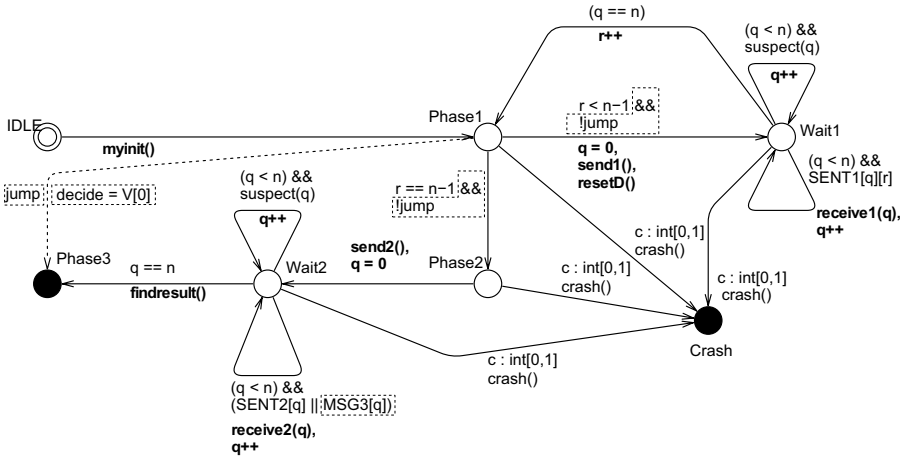

Fig. 2. The first, second and third iteration of the UPPAAL automaton.

const int n = 3;	**typedef int**[0, n-1] id_t;
bool flagTI = **false**;	domain_t MSG1[id_t][n-1][id_t];
int TI;	**bool** SENT1[id_t][id_t];
const int bottom = 0;	domain_t MSG2[id_t][id_t];
typedef int[0,n+1] domain_t;	**bool** SENT2[id_t];

Table 2
Global variable and type declarations.

int decide;	**int** c = 0;
int r = 0;	domain_t V[n];
int q = 1;	domain_t D[n];

Table 3
Local variable declarations.

1.2.1 Values and datatypes

Table 2 contains UPPAAL declarations for the global variables. The domain of agent identifiers $\{1, \ldots, n\}$ is modelled with the datatype **typedef int**$[0, n-1]$ *id_t*. It is pratical to count from 0 since we are working with arrays. The value $\perp$ is modeled by an UPPAAL constant *bottom* which equals 0. The domain of values thus becomes **typedef int**$[0, n+1]$ *domain_t*; since 0 is reserved for $\perp$.

1.2.2 Asynchronous message passing

The synchronous communication primitives of UPPAAL is not usable since we need asynchronous communication. Instead we use shared buffers. There are no test-and-set race conditions since the semantics of UPPAAL is interleaved and agents write to different places of the shared memory. We introduce the boolean arrays `SENT1[][]` and `SENT2[]`. If `SENT1[q][r]` is true, then agents `q` has made a broadcast of a message in round `r` in Phase 1. If `SENT2[q]` then agents `q` has made a broadcast of a message in Phase 2. The message is retrieved by an array lookup $v =$ `MSG1[q][r][i]`. Here v is the integer value at index `i` of the message tuple sent from agent `q` in round `r`. Similarly $v =$ `MSG2[q][i]` is a broadcast of a message with value v at index `i` from agent `q`.

1.2.3 Function definitions

See Table 4 for the source code of the UPPAAL functions needed.
Function

- `myinit()` non-deterministically selects a TI using two local variables `flagTI` and `TI`. The flag `flagTI` is set when a trusted immortal has been chosen (false, means not–yet–chosen) and `TI` holds the number of that agent. Finally vectors `V` and `D` are initialized.

```
void myinit() {
  if (flagTI == false) {
    TI = p;
    flagTI = true; }
  V[p] = p + 1;    D[p] = p + 1; }

void send1() {
  SENT1[p][r] = true;
  for (j : id_t) MSG1[p][r][j] = D[j]; }

void send2() {
  SENT2[p] = true;
  for (j : id_t) MSG2[p][j] = V[j]; }

void resetD() {
  for (i : id_t) D[i] = bottom; }

void update(id_t i) {
  for (j : id_t)
    if ((V[j] == bottom) &&
      (MSG1[i][r][j] > bottom)) {
        V[j] = MSG1[i][r][j];
        D[j] = V[j];
  }
}

void receive1(id_t i) {
  update(i);
  return;
}

void receive2(id_t i) {
  for (j : id_t)
    if (MSG2[i][j] == bottom) {
      V[j] = bottom;
    }
   return;
}

void findresult() {
  for (i : id_t) {
    if (V[i] > bottom) {
      decide = V[i];
    return;
    }
  }
}

bool suspect(id_t i) {
  return (i != p && i != TI); }

bool crash() {
  return (c && p != TI); }
```

Table 4
Local variables and functions for each agent.

- `send1()` does a Phase 1 broadcast. It copies relay vector `D` to the shared store `MSG1` and sets the sent flag in `SENT1`,
- `send2()` does Phase 2 broadcast. It copies knowledge-vector `V` to the shared store `MSG2` and sets the sent flag in `SENT2`,
- `resetD()` resets vector `D`
- `update(i)` receives a Phase 1 message and does the updates in line 10-12 of Algorithm 1
- `receive(i)` calls `update(i)`
- `receive2(i)` receives a Phase 2 message and updates V as in line 22 of the Algorithm 1
- `findresult()` does Phase 3 of the algorithm

- Function `suspect(p)` returns true if agent `p` may be suspected, i.e. is not itself or `TI`
- Function `crash(p)` returns true if agent `p` may crash, i.e. if `c` is true and `p` $\neq$ `TI`

1.3 Verification

We verify whether the model satisfies the termination and agreement property. Termination is expressed using the following invariant given as a modal logical formula:

$$\texttt{A<>}\ \texttt{forall(i : id\_t)(Agent(i).Crash or Agent(i).Phase3)} \tag{1}$$

The modality `A<>` ϕ means that *eventually* ϕ holds. That is if all possible transition sequences eventually reaches a state satisfying ϕ. Equation 1 hence describe that every agent of the system eventually crashes or reaches Phase 3. Agreement and validity is expressed by the formula below which means: There is a value y such that all agents which reach Phase 3 agree on y.

$$\begin{aligned}&\texttt{A[]}\ \texttt{exists}\ (\texttt{y : int}[1, \texttt{N} + 1])\ \texttt{forall(i : int}[0, \texttt{N} - 1]) \\ &\quad (\texttt{Agent(i).Phase3 imply Agent(i).decide} == \texttt{y})\end{aligned} \tag{2}$$

Validity is captured since the decision value y does not range over $\perp$ (or 0 in UPPAAL). Hence a value suggested by one of the agents is selected in the end.

We use model checking on a bounded number of agents and fix the number of agents to three which turns out to be enough for revealing errors. The check of (1) and (2) succeeds as expected (in total 278316 states explored, 5.2 seconds of time used).

For $n = 4$ agents we have not been able to complete the model check due to insufficiant memory (the available 3.6 gigabytes on the testsystem was not sufficient).

Another approach might include *parameterized* model checking [1] in the number of agents. At the moment it is unclear whether parameterized model checking of our correctness properties is feasible in the existing frameworks.

2 Second Iteration

In practice we may often have a situation where all agents are alive during the first two rounds of Phase 1. Hence it is possible that agreement is reached at an early point. The algorithm does not take that into account and every live agent still has to perform $n-3$ further rounds. We improve the algorithm for this situation in the following.

2.1 Design

We claim that the $n-1$ rounds needed in Algorithm 1 for reaching the agreement can be lowered to 2 in the best case. The idea is this: if all agents have a knowledge-vector of the form $(1, \cdot, \ldots, \cdot)$ then every live agent eventually decides on value 1.

We call this situation *a match*. A stop-message is sent to everybody and a jump is performed to Phase 3 immediately when a match is encountered. If agent p has received a relay vector of the form $(1, \cdot, \ldots, \cdot)$ from every agent q, then it knows that a match has occurred.

2.2 Formal Model

We now use the dashed lines and boxes containing the boolean `jump` of Figure 2. The edge from `Phase1` to `Phase3` allows a jump from Phase 1 to Phase 3 directly and the `jump` flag can instruct the automaton to jump from location `Phase1` and `Wait1` (via `Phase1`) to `Phase3`.

2.2.1 Variables

We code stop-messages by the local boolean n-array `MSG3[]`. If `MSG3[p]` is true, then agent p sends a stop-message to all. We detect matches using a local bookkeeping boolean n-array `MATCH[]`. If `MATCH[q]` is true for some agent p then V_q is of the form $(1, \cdot, \ldots, \cdot)$.

2.2.2 Function definitions

The functions are as before in Table 4 with the following exceptions (see Table 5): Function `receive1(i)` additionally check whether there is a match. If there is a match it broadcasts a stop message and sets the jump flag to true. If it receives a stop message then it sets the jump flag to true. The function `receive2(i)` sets the jump flag to true if it receives a stop-message.

```
void receive1(id_t i) {
  update(i);
  if (MSG1[i][r][0] == 1)
    MATCH[i] = true;
  if (MSG3[i] == true ||
    (forall (j : id_t) MATCH[j])) {
    q = n-1;
    jump = true;
    return; }
}
```

```
void receive2(id_t i) {
  if (MSG3[i] == true) {
    q = n-1;
    jump = true;
    return; }
```

Table 5
Updated receive functions from Table 4.

2.3 Verification

We model-check this model against the specification with $n = 3$ agents. That gives a total of 154592 states explored and a time usage of 4.7 seconds. For $n = 4$ agents we have not been able to complete the model check due to insufficient memory) and UPPAAL finds this counter example to the termination property: Assume TI = 1 and that Agent 1, 2, and 3 cooperate such that Agent 1 and 2 discover a match

and terminate (by jumping to Phase 3) while Agent 3 waits at location `Phase2`. When Agent 3 then moves to location `Wait2` a deadlock occurs. Agent 3 waits for a message from Agent 1. Since it cannot suspect Agent 1 (it is the TI) it will block until a message is sent. The problem is that Agent 1 has bypassed Phase 2, so it will never send the knowledge-vector to Agent 3 and Agent 3 waits forever.

3 Third Iteration

In the last iteration we will analyze and correct the error found in Section 2.

3.1 Design, formal model and verification

The problem with the design is the sudden termination in Phase 1 without any signaling to others. When a stop signal is received, a *stop signal should be re-sent* before jumping to Phase 3. The automaton is as in 2 where the dashed lines are included in the model. Function `receive1(i)` is as in Table 5 with the addition of the underlined code in Table 6.

```
... code as in Table 5 ...
if (MSG3[i] == true ||
  (forall (j : id_t) MATCH[j])) {
    MSG3[whoami] = true;
    q = n-1;
    jump = true;
    return; }
```

Table 6
Modified `receive1` function.

The termination, agreement and validity property are now satisfied for three agents (in total 514084 states explored in 12.3 seconds). Even though we cannot check the model for $n = 4$ (due to lack of memory) we still have an indication that the algorithm might be correct. We therefore proceed with a general correctness proof for any fixed number of agents.

4 Correctness argument for the improved CT algorithm

In this section we define what correctness is for the improved algorithm (see Algorithm 2 for the pseudo-code) and give the correctness proof for any fixed number n of agents.

Let Π as the set $\{1, \ldots, n\}$ of agents. Let $\Pi_1 \subseteq \Pi$ denote the set of agents which completed all $n-1$ rounds of Phase 1 or agents in Phase 1 that are ready to jump to Phase 3. Likewise define $\Pi_2 \subseteq \Pi$ as the set of agents that completed Phase 2 or in Phase 2 are ready to jump to Phase 3.

We adapt and reprove the following results from the original paper (see Chandra and Touegs original proof in [5]): i) Every live agent eventually reaches Phase 3, ii) For $p \in \Pi_1$: V_p has at least the knowledge of V_{TI}, iii) For $p \in \Pi_2$: V_{TI} has the same knowledge as V_p. The new variable A_p is a set which contains the agents that have a knowledge-vector of the form $(1, \cdot, \ldots, \cdot)$. The following lemma is added: If $|A_p| = n$ for an agent p then V_q is on the form $(1, \cdot, \ldots, \cdot)$ for every live agent q. As a consequence every live agent will decide on the value 1.

Algorithm 2 Distributed Agreement, improved effiency.

Require: An agent number $p \in \{1, \ldots, n\}$
Ensure: A final decision value in the interval $1, \ldots, n$

$V_p \leftarrow \overline{\perp}$, $V_p(p) \leftarrow p$, $\Delta_p \leftarrow V_p, A_p \leftarrow \emptyset$

Phase 1:
for all $r_p \leftarrow 1$ to $n-1$ **do**
  send message $\mathsf{Phase1}(p, r_p, \Delta_p)$ to everyone
  $\Delta_p \leftarrow \overline{\perp}$
  for all $1 \leq q \leq n$
    block until
      receive $m = \mathsf{Phase1}(q, r, \Delta)$
      if $V_p(q) = \perp \wedge \Delta(q) \neq \perp$ **then**
        $V_p(q) \leftarrow \Delta(q)$
        $\Delta_p(q) \leftarrow \Delta(q)$
      if $\Delta(1) \neq \perp$ **then** $A_p \leftarrow A_p \cup \{q\}$
      if $m = \mathsf{Phase3} \vee |A_p| = n$ **then**
      send message Phase3 to everyone and **goto** Phase 3
    or suspect q

Phase 2:
send message $\mathsf{Phase2}(V_p)$ to everyone
for all $1 \leq q \leq n$ **do**
  block until
    receive $m = \mathsf{Phase2}(V)$
    if $m = \mathsf{Phase3}$ **then goto** Phase 3.
    if $V(q) = \perp$ **then** $V_p(q) \leftarrow \perp$
  or suspect q

Phase 3: decide $= \min\{q \mid V_p(q) \neq \perp\}$

Lemma 4.1 (Termination) *Every live agent will eventually reach Phase 3. When run on n agents it terminates after n rounds in the worst case and 2 rounds in the best case.*

Proof. The only way a live agent p can be prevented from reaching Phase 3 is if it is blocked at the receive/suspect loop at Phase 1 or Phase 2. This happens when

(i) p is waiting to receive a message from q which never arrives and

(ii) p cannot suspect agent q.

Let q be such an agent. Since q cannot be suspected it must be the trusted immortal (it cannot be itself since agents can always receive their own value).

Claim: The trusted immortal cannot deadlock.

This follows because TI can always complete Phase 1 and Phase 2 unhindered. In both phases it may simply suspect every other agent and receive its own relayed messages.

Now, p waits for q and q will not block. This means that q will send the message p is waiting for. That might either be a Phase1, Phase2 or Phase 3 message. Due to the fairness assumption (sent messages should eventually be received) p will continue.

In the best case a match is discovered after two rounds. The worst case running time is achieved when all iterations of Phase 1 and Phase 2 have to be performed. That gives n rounds in total. □

Lemma 4.2 *If $|A_p| = n$ for an agent p then V_q is on the form $(1, \cdot, \ldots, \cdot)$ for every live agent q. As a consequence every live agent will decide on the value 1.*

Proof. Assume that p is an agent for which $|A_p| = n$. This means that p has received relay vectors on the form $(1, \cdot, \ldots, \cdot)$ from every other agent. Since relays only contain the broadcast of learned values we know that $V_q(1)$ also equals 1 for every other agents q. By the termination lemma above and the code of Phase 3 (which returns the first non-bottom value of the knowledge-vector) it follows that every agent which reaches Phase 3 decides on value 1. □

Definition 4.3 Define an ordering called matched-less-than-or-equal-to $\leq_m$ on n-vectors by $V_1 \leq_m V_2$ if either $V_1(1) = V_2(1) = 1$ or $V_1 \leq V_2$. When writing $V_1 =_m V_2$ this means that $V_1 \leq_m V_2$ and also $V_2 \leq_m V_1$.

Lemma 4.4 *For all $p \in \Pi_1$, $V_{\mathrm{TI}} \leq_m V_p$ at the end of Phase 1.*

Proof. Suppose that $V_{\mathrm{TI}}(q) = q$, $1 \leq q \leq n$ at the end of Phase 1. For every $p \in \Pi_1$ we must show that by the end of Phase 1: $V_p(q) = q$ (by completion of $n-1$ rounds) or that there was a match.

If p is at the end of Phase 1 due to a match we may apply Lemma 4.2 which gives that $V_{\mathrm{TI}}(1) = 1$ and $V_p(1) = 1$ so $V_{\mathrm{TI}} \leq_m V_p$ as wanted.

So now assume that p is at the end of Phase 2 and not because of a match. If $p = \mathrm{TI}$ then clearly $V_{\mathrm{TI}} \leq_m V_p$. So assume $p \neq \mathrm{TI}$ and let r be the first round in which TI received value q (if $\mathrm{TI} = q$ let r be 0). There must have been a relay of that value so $\Delta_{\mathrm{TI}}(q) = q$ at the end of round r. Either r is the last possible round or it is not:

(i) **Not in Last round**, so $r \leq n-2$. In round $r+1$, TI relays value q by sending

the message $\mathsf{Phase1}(\mathrm{TI}, r+1, \Delta_{\mathrm{TI}})$ with $\Delta_{\mathrm{TI}}(q) = q$ to all. Agent p is forced to receive the value $\mathsf{Phase1}(\mathrm{TI}, r+1, \Delta_{\mathrm{TI}})$ in round $r+1$ from TI. By the update in line 10-12 of the algorithm it follows that p sets $V_p(q)$ to q by the end of round $r+1$.

(ii) **In last round**, so $r = n-1$. Here TI received q in round $n-1$ for the first time. Since each agent relays q (using the relay vector) at most once, it follows that q was relayed by all $n-1$ agents in $\Pi \setminus \{\mathrm{TI}\}$, including p, before being received by TI. Since p sets $V_p(q) = q$ before relaying q, it follows that $V_p(q) = q$ at the end of Phase 1.

□

Lemma 4.5 *For all $p \in \Pi_2$, $V_{\mathrm{TI}} =_m V_p$ at the end of Phase 2.*

Proof. Let $p \in \Pi_2$ and a given $q \in \Pi$. If p is at the end of Phase 2 and there is a match then from Lemma 4.2 we know that $V_{\mathrm{TI}} =_m V_p$.

If this is not the case we will show that $V_p(q) = V_{\mathrm{TI}}(q)$ by looking at two cases:

- $V_{\mathrm{TI}}(q) = q$ at the end of Phase 1. From Lemma 4.4, for all agents $p' \in \Pi_1$ (including p and TI), $V_{p'}(q) = q$ at the end of Phase 1. Thus all vectors V which are sent in Phase 2 have the property $V(q) = q$. Hence, both $V_p(q)$ and $V_{\mathrm{TI}}(q)$ stay equal to q through Phase 2.
- $V_{\mathrm{TI}}(q) = \perp$ at the end of Phase 1. Since TI cannot be suspected, p waits for and receives V_{TI} in Phase 2. Since $V_{\mathrm{TI}}(q) = \perp$ then p sets $V_p(q) = \perp$ at the end of Phase 2.

As a result it holds that TI and p have the same knowledge and the same $\perp$'s at each index. This proves the claim. □

Corollary 4.6 (Agreement) *No two agents decide differently.*

Proof. Follows from lemma 4.5 and the function `decide`. □

Lemma 4.7 *For all $p \in \Pi_2$ it holds that $V_p(\mathrm{TI}) = \mathrm{TI}$ at the end of Phase 2.*

Proof. Clearly $V_{\mathrm{TI}}(\mathrm{TI}) = \mathrm{TI}$. By Lemma 4.4 we have that $V_q(\mathrm{TI}) = \mathrm{TI}$ for all $q \in \Pi_1$ at the end of Phase 1. So no agent sends V with $V(\mathrm{TI}) = \perp$ under Phase 2. From the code of Phase 2 in the algorithm it is clear that $V_p(\mathrm{TI}) = \mathrm{TI}$ for all $p \in \Pi_2$ at the end of Phase 2. □

Lemma 4.8 (Validity) *The improved algorithm satisfies validity.*

Proof. From the pseudocode of the algorithm it is clear that only values origianally suggested by agents are relayed and moved around in knowledge vectors. Validity thus boils down to the proving that agents cannot agree on $\perp$.

Lemma 4.5 gives that $V_{\mathrm{TI}} =_m V_p$ for all $p \in \Pi_2$ at the end of Phase 2. Lemma 4.7 gives that there at least is one entry in V_p (namely at index TI) which is non $\perp$. Hence $\perp$ cannot be selected which gives the desired. □

Theorem 4.9 *The improved Algorithm 2 satisfies the termination, agreement and validity property. When run with n agents it terminates after n rounds worst case and 2 rounds in the best case.*

Proof. Lemma 4.1 and 4.6 and 4.8 proves the properties. □

5 Conclusion and Future Work

We applied formal model–driven development to a realistic distributed algorithm. As a result we found an improvement of an existing agreement algorithm and proved it correct. The improved algorithm may significantly speed up the process of reaching agreement in practical situations. Model–driven development is useful for the design of new distributed algorithms for a number of reasons:

(i) High level models are easier to work with than source code.

(ii) Insights gained during modeling and through simulations help the intuition in both designs and possible correctness arguments.

(iii) Design errors which are normally discovered at proof stage are often found faster through verification.

(iv) Automatic construction of counter examples is of great help.

The state space explosion problem often makes it hard to verify systems even with a small number of agents. Nevertheless, counter examples often turn up for a small number of agents in practice.

Future work involves qualitative studies of agreement algorithms. If we limit the communication time we may ask: "How long time does it take to reach an agreement?". UPPAAL is well suited for analysis of such questions. An approach very different from ours is found in [13] where correct agreement algorithms are found automatically. This seems to be an interesting direction of future research in distributed algorithms too.

Acknowledgement

I thank Jiří Srba for his comments.

References

[1] Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine, *Monotonic Abstraction in Parameterized Verification*. Invited Contribution in proceedings of RP 2008. 2nd Workshop on Reachability Problems, Liverpool, UK, 2008.

[2] G. Behrmann and A. David and K.G. Larsen, *A Tutorial on* Uppaal, In Formal Methods for the Design of Real-Time Systems., number 2185 in LNCS, 2004.

[3] Ben-Ari, Mordechai, *Principles of the Spin Model Checker*. ISBN 978-1-84628-769-5, XVI, 2008.

[4] Mike Burrows, *The Chubby lock service for loosely-coupled distributed systems*. Proceedings OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, USA, November, 2006.

[5] Tushar Deepak Chandra and Sam Toueg, *Unreliable Failure Detectors for Reliable Distributed Systems*, J. ACM, Volume 43, number 2, pages 225–267, 1996.

[6] Adrian Francalanza and Matthew Hennessy, *A Fault Tolerance Bisimulation Proof for Consensus (Extended Abstract)*, Proceedings of ESOP'07: 16th European Symposium on Programming, pages 395–410, March 2007.

[7] Wilhelm Hasselbring, *Programming languages and systems for prototyping concurrent applications*, ACM Comput. Surv., Volume 32, Number 1, pages 43–79, 2000.

[8] Morten Kühnrich and Uwe Nestmann, *On Process-Algebraic Proof Methods for Fault Tolerant Distributed Systems*. To appear at proceedings at FMOODS 2009: 11th Formal Methods for Open Object-Based Distributed Systems as a part of IFIP: international conference on Formal Techniques for Distributed Systems. 2009.

[9] Nancy A. Lynch, *Distributed Algorithms*, ISBN 1558603484, San Francisco, CA, USA, 1996.

[10] Stephen J. Mellor and Anthony N. Clark and Takao Futagami, *Guest Editors' Introduction: Model-Driven Development*, IEEE Software, Volume 20, Number 5, pages 14–18, 2003.

[11] Uwe Nestmann and Rachele Fuzzati, *Unreliable Failure Detectors via Operational Semantics*, In procedings of Advances in Computing Science, ASIAN 2003: Progamming Languages and Distributed Computation, 8th Asian Computing Science Conference Mumbai, India, December 10-12, LNCS, Volume 2896, pages 54–71, 2003.

[12] Tatsuhiro Tsuchiya and André Schiper, *Using Bounded Model Checking to Verify Consensus Algorithms*, In proceedings of Distributed Computing: 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, LNCS, Volume 5218, pages 466–480, 2008.

[13] Piotr Zielinski, *Automatic Verification and Discovery of Byzantine Consensus Protocols*, In proceedings of DSN '07: The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Edinburgh, UK. June 25 - June 28, 2007,